Software Development for High Performance Computing

Teodor A. Rus and Arthur C. Fleck
Department of Computer Science
The University of Iowa
Iowa City, Iowa 52242
rus,fleck@herky.cs.uiowa.edu

July 14, 1995

Contents

Abstract

Gigabytes and teraflops are necessary but not sufficient to achieve the goals established by the Strategic Implementation Plan: America in the Age of Information issued by the Committee on Information and Communications R&D. We believe that the most promising direction for near term progress is in improving control over the computing power of the high performance computer systems. Our paper identifies what we regard as the most important challenges and goals in this area and defines the activities and efforts that we see as most important for progress.

1 Goals

Gigabytes and teraflops are necessary but not sufficient to achieve the goals established by the Strategic Implementation Plan: America in the Age of Information issued by the Committee on Information and Communications R&D. Software provides the vital bridge between computing hardware and the needs of any particular application. A primary goal of software support must be to provide end users with flexible, adaptable tools that are well suited to their needs. With the growing diversity and complexity of applications, this implies that an abundance of software tools will be required. To maximize productivity, it is important that the tools be formulated in such a way that the end users can easily tailor the function to aspects peculiar to their application area. This primary goal can be achieved only if the developers who produce the software tools for end users themselves have powerful general systems tools to support their work. It is essential to recognize that the complex software required in the future can only be achieved by developing first a suitable modeling of the problem solving with the computer and by a mathematical methodology for software development. In addition, the usage of this software requires us to create the culture of parallel computing. For that we need a critical mass of programmers capable of effectively using the existing parallel machines. This necessitates appropriate education in parallel computing at both the graduate and undergraduate levels.

1.1 Limitations of current software technology

Major difficulties in using currently available High Performance Computer Systems, (HPCS), result from the two equally challenging limitations of the current technology: lacking the mathematical models and solution algorithms to solve grand challenge problems and (when these models and algorithms are available) lacking the software that would allow computer users to control the computation power provided by the HPCS according to their own computation needs. To understand this situation we have to look at the historical development of the current paradigms for using computers to solve problems.

These paradigms are based on a concept of *control flow*, which is an abstraction of the instruction cycle of the sequential machines. The computation models used as foundations for problem solving with a computer require the encoding of the solution algorithms as stored programs and control tools for starting, controlling, and terminating program execution. That is, the major software tools supporting problem solving with a sequential machine are the *compiler* and the *operating system*. The compiler allows the programmers to develop their programs using high-level language notations called *programming languages* rather than the binary notation used by the computer to express and execute the stored programs. The operating system controls the process of program execution allowing efficient use of the computer resources, memory and input-output devices.

Both the compiler and the operating system design and implementation allowing convenient and efficient use of the computer are still the major tools of the current system software technology. However, neither the compiler nor the operating system design and implementation have matured into a mathematical methodology. Hence, they are difficult to adapt to the new problem domains they control and service. In addition, both the compiler and the operating system still provide a frozen computing environment while the application software they serve is dynamic and evolves with the problem domain. Hence, the control-flow is probably the single most important concept of the current software technology because it is the basis of the current software tools for problem solving with sequential machines. But this is not a unifying concept for sequential and parallel computing [10] and therefore it is probably also the most important impediment in the development of a new software technology suitable for problem solving with HPCS.

1.2 Goals of the future software development

In order to overcome current limitations and develop a software technology suitable for problem solving with HPCS, it is essential to recognize that software complexity grows with both hardware sophistication and application domain and neither the new hardware nor the new application domains can be modeled by means of control-flow. Therefore, the complex software required in the future can only be achieved by developing a mathematical methodology for complex problem solving that unifies parallel and sequential program development and integrates software tools used for this purpose. The core of the computing system is the hardware itself, and the outermost layer is the end user software. But there are many intermediate layers including operating systems, language compilers, networking protocols, etc. Only when each layer of this hierarchy of computing systems is appropriately constructed can further developments be effectively completed. Although this structuring is visible in the current software, it grew up in an ad hoc manner and lacks the capability to integrate the layers and to unify the paradigms for program development. That is, the fundamental problem of the current software technology is actually generated by the contradiction between the current system software and the application software. This contradiction is particularly visible in the context of the HPCS whose computation paradigms are not necessarily sequential and which are controlled by a system software derived from sequential machines. This contradiction is aggravated by the tremendous development of the HPCS, both as a means to advance their mission and as a national resource to enhance American productivity and competitiveness [17].

The resolution of this conflict cannot be obtained by patching the conventional system software, rather investment in the development of a new system software suitable for the HPCS is necessary for that. As one can see from the analysis of the current situation presented in [17], the time for making such an investment has arrived. The research areas to be emphasized are the discovery of new and unifying concepts for program development and implementation mechanisms, and the creation of a mathematical methodology for system software design and implementation. Scalability of the system software tools and the incremental nature of the application software developed with them are probably the major characteristics of these models.

The goals we stress in this paper relate to the development of tools that support each layer of the overall software system. These layers emphasize different levels of abstraction, with inner layers being closely intertwined with the hardware and outer layers being strongly oriented to the problem domain. To serve effectively, the software must strive to match the appropriate level of abstraction. We advocate software design that permits the end users to focus their attention on the problem domain, and leaves the mapping of the computation into whatever hardware platform is being used to the lower levels of the system that are optimized for that particular hardware.

2 Opportunities

There have been extraordinary developments in software systems in recent years. New programming paradigms have been introduced: functional programming, logic programming, and object-oriented programming. The understanding and practice of each of these areas has advanced rapidly. Also the development of formal specification of software systems has progressed enormously.

2.1 Abstractions provide great opportunities

Through the development of specification languages that can be executed to achieve rapid prototyping, and the declarative view (i.e., functional and logic programming), specification and programming have grown very close together. In fact, recently even text documents have been growing "smarter". The exploding popularity of presenting documents on the Internet through the HTML language is because this permits us to incorporate some elements of computation directly into our text. When one activates a link in a Web document, a computation closely analogous to a remote procedure call is performed. Of course the procedure calls are very inflexible in the current setting, but it does not take much imagination to foresee many other possibilities. Yet, paradoxically, none of these advances can really take advantage of the computation power provided by the new parallel machines, and their migration towards these machines is too slow with respect to the urgency of the goals we discuss here. We believe that this is due to the fact that methodology and implementation tools for parallel machines has not kept pace with the rapid development of formal specification mechanisms. Therefore, the discovery of new implementation mechanisms, naturally suited to the formal specification tools used to develop the new paradigms of computation, will create the opportunity to merge elements from these diverse approaches to substantially benefit the construction of software tools for HPCS at all levels. We advocate this process in the context of developing new languages and compilers that support incremental development of programs and allow programmers to control the granularity of the processes executing their programs and effectively utilize the computation power provided by the hardware.

2.2 Why do we need new compilers?

To understand the advantages provided by our approach we need to start by summarizing the conventional approach of compilation. Conventional technology for compiler construction evolved from the need to release the programmer of the burden of writing machine-language programs. That is, a conventional compiler allows programmers to develop their programs using a notation closer to the problem domain while the computer is designed to execute programs written in a binary machine language. This technology is not founded upon a formal concept of a programming language and is not based on a mathematical algorithm that models a compiler. Rather, it consists of a series of well understood transformations performed on the notations used by programmers to express their algorithms mapping them into the binary notations that represent machine language programs. The lack of a mathematical model for the compiler makes it difficult to integrate these transformations into a software artifact that can be proven correct and is understandable to its user, the programmer. The side effect is that the compiler implementation is a difficult task and the correctness of the commercial compilers is usually not proven mathematically. This situation is typical for the current software and has led to profound problems in the software industry [19].

In addition, a compiler may incorporate assumptions on its source and target languages that are not necessarily acceptable for another compiler that has the same source and target languages. The consequence is that programs are not portable between platforms of machines nor between generations of languages. On the other hand while a compiler freezes the notation that programmers can use to develop their programs, the problem domain evolves and requires extensions that are not supported by the compiler. The programmer must then use old and sometimes inappropriate tools to solve new problems. Consequently the computation power of the new machines cannot actually be used and the efficacy of problem solving is reduced. The best example illustrating this conclusion is provided by the need to exploit parallel machines using sequential languages. This state of the art leads to the paradoxical situation where the compiler developed as a tool to make programming easier becomes a burden for the programming activity.

2.3 New methodology for compiler design

To avoid the consequences of the situation created by the historical evolution of the conventional compilers we need to develop a new methodology for language design and implementation that while accommodating the current programming tools would allow programmers to manufacture their own languages and compilers adapted to their own machines and problem domains. Such a methodology becomes feasible if it is based on mathematical concepts of programming language and programming language translation that are independent of the computer and computer user and could be easily mastered by the programmers. Since there is already a rather long history of using the universal algebra as a framework for language specification [11] and some successful experiments on compiler modeling by algorithms for homomorphism computation [8, 21] we are seeking the new methodology for language processing in the framework of universal algebras.

The difficulty in using the algebraic methodology for the development of a new technology for language processing resides in the way programming languages evolved as "notations with which people can communicate algorithms to computers and to one another" [2] and the manner in which algebraic mechanisms have been used to specify such a notation. In other words, while the concept of a programming language evolved as a notation of well understood but unspecified computations, the compiler design is based on formal specifications of both the computations expressed by a programming language and the notation used to express these computations. The notation part of the programming language, henceforth called the *syntax*, was precisely described within the framework of formal languages [15]. The computations expressed by the syntax, henceforth called the *semantics*, have been formalized using the domain theory [14] which is a different framework. These formalizations of syntax and semantics of a programming languages allowed us to understand the compilation process and to develop current technology for programming language processing but did not mature into a formal concept of a programming language and into a mathematical model of a compiler. The cause of this state of the art might be found in the lack of a mathematical framework naturally integrating the syntax, the semantics, and the compilation process.

Recently the universal algebra is used as a specification mechanism that integrates semantics and syntax [24] but the compilation process is still controlled by automata capable to recognize the syntax of programming languages. However, we have shown [21] that universal algebra can be used as a framework that integrates programming languages and their processors and have developed a *technology for implementing computing systems*, TICS system, where a compiler algorithm is developed on the basis of homomorphism computation rather than on automata theory. A compiler developed with TICS system allows programmers to develop their programs incrementally and thus frees program development from the restriction of *monolithic programming* imposed by context-free grammars. Correctness of a compiler generated by TICS is mathematically established and program portability between languages and machines is assured by language to language translators rather then by compatibility restrictions. Our experience shows that this paradigm of compilation supports automatic generation of production quality compilers and compiler customization by the programmer thus adapting the software environment to the problem domain and hardware platform.

3 Ongoing activities

Three categories of on-going activity provide direct contributions to the objectives and goals of the Strategic Implementation Plan: America in the Age of Information and consequently warrant increased support. Namely, these activity areas are: research on revolutionary approaches to parallel software development, educational programs providing extensive hands-on experience with parallel environments, and promoting the international cooperation within the technical communities pursuing solutions to these problems.

3.1 Promoting revolutionary approaches

Any decision regarding the national and international research support targeted to contribute to the accomplishment of the goals established by the Strategic Implementation Plan: America in the Age of Information must be based on a realistic analysis of the current situation in software development. Important considerations in such a decision should be both the continuity of the on-going research, which assures the reliability of the software industry, and the courage to launch new projects that lead to the goals, which assures the progress and competitiveness of the software industry.

We identify three major areas of on-going research activities that should be continued but better guided towards these goals. They are methodology and foundations, information dissemination, and support for revolutionary discovery.

Much helpful research has been done on formal foundations for software development. However, experience shows that most of the research directed towards the mathematical foundation of computer artifacts sooner or later degenerates into pure mathematics. Therefore, concerted national and international efforts must be supported to focus the research on mathematical foundations *to develop software technology on a mathematical basis rather than develop a new mathematics suggested by software objects developed in an ad-hoc manner.* We believe that a relevant guiding formalism is a essential basis for managing the complexity of high performance systems.

Important contributions have been made in high performance computing. However, we take the growing discrepency between the capacity of hardware and software for high performance computing to be a signal that evolution from sequential software technology is an inadequate approach for future (or even today's) systems. The mismatch between hardware capacity and software facilities is so serious as to even threaten the advances in hardware as evidenced by the number of companies producing parallel computing hardware that have gone bankrupt. Both our funding agencies, and our outlets for dissemination of ideas need to be more favorably disposed to exploring revolutionary new approaches to software development for high performance computing. While continuity with the advances that have been made must be maintained, radical new approaches must also be fostered.

Technically focussed conferences provide a vital means for the timely dissementation of new results. However, most often national and international meetings destined to report and share scientific results in the area of software development also have the goal of making a profit for the organization arranging the meeting. Therefore, the organizers select exciting sites (usually expensive resorts) and invite highly sought after specialists. This makes these meetings excessively expensive and not available to students and young researchers. But the discovery of new revolutionary approaches in software technology requires the essential involvement of the young generation. This suggests that using public funds to promote more broadly accessible conferences would be a valuable aid to attracting new workers to the area.

3.2 Education for parallel computing

In order to keep our country at the forefront of this area of activity, we need to provide the educational opportunities that will sustain a viable group with suitable technical skills. This will require a major investment in new curricula development that emphasizes hands-on approaches to teaching parallel computation. We need to develop and disseminate the theoretical basis of parallel computation in conjunction with practical experience, and to permeate the hands-on experience with the development of intuition into parallel computing.

There is a sustained trend on developing means and tools for education on parallel computing [18]. However, when carefully examined we may observe that while many courses we teach bear the attributes "parallel computing" and "high performance" in their titles, very few of them do really provide hands-on experience with HPCS. This is probably due to the fact that the existent equipment in computing laboratories is mainly based on sequential systems. On the other hand the new parallel machines are still expensive and do not generate enough profit and thus cannot be routinely available at educational institutions. It is true that NSF supercomputing centers are organized in the academic environments and offer computation time. But it is also true that they are inconvenient for small teaching jobs, are not provided with educational tools that would facilitate hands-on experience and development of intuition of parallelism and therefore while they may provide for advanced research they serve educational purposes poorly. Consequently, the critical mass of programmers for parallel machines required in order to sustain the endeavor of producing parallel machines cannot be created and the endeavor itself is threatened.

3.3 International cooperation

Recognizing the importance of new methodology for software development we created the international movement of using *Algebraic Methodology* for the development of *Software Technology*, AMAST. AMAST was initiated in 1989 with its first meeting held in Iowa City, May 21-25. It was joined by many well-known professionals interested in using algebraic methodology for the development of software technology. We stress that the AMAST movement is much more than wishful thinking, it is a reality that can be measured by the increasing number of people joining it and by accomplishments that show that it does realize its major goal: *the mapping of algebraic methodology into software technology.* The major accomplishments of AMAST are:

- AMAST has held 3 international conferences and two international workshops. The amount and the quality of the papers submitted, as well as the increasing number of attendees, made us realize that we should initiate a yearly international conference and international workshops, starting in 1995.

- AMAST has so far published its results in three full issues of Theoretical Computer Science as well as three proceedings of the papers presented at its conferences.

- The AMAST Series in Computing has been created and has published three monographs [4, 11, 21] reporting essential results towards the development of a mathematical methodology for software technology.

- At each AMAST meeting we have organized demonstrations of real-life systems designed and implemented following the rules of the mathematical methodology we are developing. In addition, starting with our 1993 meeting we dedicate one day to the mathematical education of the software engineer. On this day we teach the methodology of applying mathematical results to the design and implementation of software systems.

Due to the rate of growth recently faced by the AMAST movement we feel that we need to enlarge USA involvement both with respect to its research themes and its geographical distribution. This implies carefully balancing the AMAST activity and a firm movement toward an international cooperation built around the USA – Europe enterprise generated by AMAST, which in many ways is unique. Therefore we feel that this should no longer evolve only as a grass-roots movement, but instead should be anchored in the general concern supported by Committee on Information and Communications R&D (CIC) of the National Science and Technology Council (NSTC), whose objectives match the objectives of the AMAST.

4 New efforts

As preceding comments have indicated, there are several aspects that we believe are deserving of a national emphasis. A diverse collection of new ideas in programming languages has been developed in recent years and it is important to recognize and apply those that are useful to this context.

4.1 Building successes on accomplishments

Despite an explosive variety and volume of new programming language ideas, no compelling language for parallel programming has appeared. Declarative languages focus on what *results* a computation is to produce rather than the computation itself. This provides a means of expression that *permits* parallelism, but does not explicitly express it. A language is needed that incorporates the two missing concepts from current programming languages, *the process* and *the file*, as central facilities, and that provides a higher-level means of writing parallel programs by allowing the programmer to develop the intuition of parallel computation.

Object-oriented programming is one of the most promising paradigms in this regard. With its message-passing discipline, object-oriented programming has great potential for the description of many aspects of process creation and interaction, and some initial attempts at this approach have been made. It will be important to pursue this in the context of a strong typing system to enhance the reliability of programs. Investigations of coherent and orderly alternatives in this direction should be aggressively encouraged as the development of a salutary language in which to write parallel programs would have a huge impact.

At the same time merging useful ideas from other programming paradigms should not be overlooked. For instance, pursuit of the integration of logic programming with object-oriented programming has been active recently, and this could be very helpful approach to obtaining a specification of object classes that would support essential interface abstraction. The exaggerated difficulty of ensuring correctness of parallel programs implies that issues of specification and verification need to be given recognition and support in the language design right from the outset.

4.2 Towards improved education for parallel computation

Current experience with parallel program development shows that it is essential for computer users to understand that *many implicit activities related to the development and execution of programs must be explicitly carried out by programmers when developing parallel programs.* Such activities are: creating a computation model in which intelligent computing agents perform simultaneously portions of the program; controlling the computation agents while they execute their portions of a program; splitting the computation of the program into subcomputations and allocating these subcomputations to the intelligent computing agents. The major difficulty in implementing parallel algorithms is generated by the lack of intuition and software support for these activities. Since the process that executes a sequential program is unique for each program run on a sequential machine, it can be implicitly created and managed by the operating system, and thus the programmer needs not be aware of its existence. Consequently, neither the intuition of parallel processes populating the operating system, nor the language support to manage these processes are supported by sequential programming languages.

The critical mass of programmers able to handle existing parallel machines and to sustain the future developments as suggested by the Strategic Implementation Plan: America in the Age of Information can be created in the American Schools. For that to happens, HPCS must be provided with open-ended programming environments that allows the programmer not only to use programming tools but also to create new tools and to update the environment with the tools he or she needs.

4.3 Open-ended programming environments

Programming environments for parallel machines in use today are built on previous experience with sequential program development. They can be classified as:

- Implicit parallel programming [1, 16, 6, 25], where programmers develop sequential programs and compilers discover parallelism and create processes.

- Explicit parallel programming, where programmers use sequential languages paired with a coordination language for process creation and management. The coordination language is either supported by the operating system and compiler [7, 12, 13], or is independent [9, 5] and thus provides for parallel program portability.

- Machine-dependent parallel programming, developed on processor arrays and heterogeneous networks, much like sequential programming was developed [20, 3].

None of these parallel programming environments satisfies the essential condition for a true education in parallel computing: *the development of the intuition for parallel programming as a distinct computational phenomenon.* However, all of them could contribute to this intuition by allowing us to develop hands-on experience with the currently available machines. Programming environments for HPCS is a crucial topics for future software technology because a suitable environment provides for both research and education and may have good feedback on the technology itself. But we stress here that with a parallel programming environment many implicit activities in the conventional environments must be explicitly carried out and therefore the concept itself must be carefully designed. Building upon the conventional interactive environments, we can introduce a unifying concept of programming environment as a tuple $\mathbf{Env} = \langle Tools, Language, Interpreter\rangle$ where $Tools$ is a collection of objects (data and programs) used for program development and execution, $Language$ allows the programmer to express requests to the environment, and $Interpreter$ maps $expressions \in Language$ into streams of control on the underlying machine.

4.4 A new logic for problem solving

The programmer's dream is to discover the ideal programming environment that supports problem solving without writing programs. The process programming language [23] may be a step towards this goal. However, for this dream to become true a *new logic for problem solving with a computer* must be developed first. The essential characteristics of this logic are:

- The computation units in this logic are processes rather than instructions executed by the computer. The instruction cycle encapsulated in the hardware of a computer system is such a process. But the process in this context is viewed as the computation unit that reflects the programmer's logic for problem solving rather than the logic of the computation as performed by the hardware.

- The algorithm describing a computation shows the flow of control among processes performing various computation tasks that make up the algorithm rather than control flow of computer instructions.

- The program in memory and the program counter in the processor are replaced by environment integrated tools and the interpreter executing algorithms whose primitive operations are processes of executing tools of the environment rather than machine instructions.

Steps towards the new logic for problem solving have already been made by the fifth generation machines and related systems [22], where computation units are the operations of logic deductions, by the abstraction management by definition, declaration, and application, as implemented by the **Hydra** [26] operating system, and by the software environments suggested by the *Arcadia* corporation [23]. We notice however that the previous research on programming environments tries to accomplish the goals raised by the programmer's dream within the framework provided by the ad hoc methodology for problem solving based on conventional programming. To be successful we advocate the development of a mathematical methodology for problem solving and a programming environment design based on a new logic for problem solving, where the solution of a problem is a linguistic expression of the solutions of other simpler problems.

5 Challenges for software development

We close by summarizing the points that constitute the basic challenges which are most deserving of serious attention in the immediate future:

1. create the new logic for problem solving based on open-ended programming environments for high performance computer systems

2. develop a formal methodology that guides us toward the construction of correct and portable parallel programs, and adopt an openness to radical and innovative alternatives

3. design a programming language that incorporates a unifying intuitive model of parallel computation, and which provides a coherent vehicle for the natural description of parallel programs

4. devise and construct software tools that resonate with the methodology and facilitate a flexible, supportive environment

5. introduce widely available, substantial educational opportunities in parallel programming that will create a pool of individuals with the experience and intuition necessary to work effectively in this setting.

References

[1] J.C. Adams, W.S. Brainerd, Martin J.T., B.T. Smith, and J.L. Wagner. *Fortran 90 Handbook, Complete ANSI/ISO Reference*. Intertext Publications. McGraw-Hill Book Company, 1992.

[2] A.V. Aho and J.D. Ullman. *Principles of Compiler Design*. Addison–Wesley Publishing Company, Reading, MA, 1977.

[3] G.S. Almasi and A. Gottlieb. *Highly Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., Redwood, CA, second edition edition, 1994.

[4] A. Arnold, D. Bégay, and Curbillé P. *Construction and Analysis of Transition Systems with MEC*. AMAST Series in Computing, Vol. 3. World Scientific, 1994.

[5] Scientific Computing Associates. *C-Linda Reference Manual*. Scientific Computing Associates, Inc., New Haven, Connecticut, 1990.

[6] U. Banerjee. *Dependency analysis for supercomputing*. Kluwer Academic Publishing, Norwell, MA, 1988.

[7] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso. *Programming Under Mach*. Addison-Wesley, Reading, MA, 1993.

[8] R.M. Burstall and P.J. Landin. Programs and their proofs: an algebraic approach. *Machine Intelligence*, 4:17–43, 1969.

[9] N. Carriero and D. Gelernter. *How to write parallel programs: A first course*. MIT Press, Cambridge, MA, 1990.

[10] K.M. Chandy and J. Mishra. *Parallel Program Design – A Foundation –*. Addison-Wesley Publishing Company, Reading, MA, 1988.

[11] I. Classen, H. Ehrig, and D. Wolz. *Algebraic Specification Techniques and Tools for Software Development*. AMAST Series in Computing, Vol. 2. World Scientific, 1993.

[12] ENCORE Computer Corporation. *Encore Parallel Threads*, 724-06210, rev. a edition, May 1988.

[13] I. Foster. *Designing and Building Parallel Programs – Concepts and Tools for Parallel Software Engineering –*. Addison-Wesley Publishing Company, Reading, MA, 1995.

[14] C.A. Gunter and D.S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Sciencei – Volume B: Formal Models and Semantics*, pages 633–674. The MIT Press/Elsevier Science Publishers, 1990.

[15] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley Publishing Company, Readings, MA, 1979.

[16] Argonne National Laboratory. Using the alliant fx/8. Technical Report Technical Memorandum Number 69, Rev. 1, Argonne National Laboratory, Mathematics and Computer Science Division, September 1986.

[17] P. Messina and T. Sterling, editors. *System Software and Tools for High Performance Computing Environments*. SIAM, Society for Industrial and Applied Mathematics, Philadelphia, 1993.

[18] R. Miller. The status of parallel processing education. *IEEE Computer*, pages 40–43, August 1994.

[19] P.G. Neumann. *Computer Related Risks*. Addison-Wesley Publishing Company, Reading, MA, 1995.

[20] M.J. Quinn. *Parallel Computing – Theory and Practice –*. McGraw-Hill Series in Computer Science. McGraw-Hill, 1994.

[21] T. Rus. Algebraic construction of compilers. *Theoretical Computer Science*, 90:271–308, 1991.

[22] E. Shapiro, editor. *Concurrent Prolog: collected papers*. MIT Press, 1987.

[23] R.N. Taylor, F.C. Belz, L.A. Clarke, L. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf, and M. Young. Foundation for the arcadia environment architecture. In *Proceedings of SIGSOFT'88: The Third Symposium on Software Development Environments*, pages 1–13, November 1988.

[24] M. Wirsing. Algebraic specification. In managing editor J. van Leeuwen, editor, *Handbook of Theoretical Computer Science – Volume B: Formal Models and Semantics –*. The MIT Press/Elesevier, 1990.

[25] M.J. Wolfe. *Optimizing supercompilers for supercomputers*. Pitman Publishing, MIT Press, Cambridge MA, 1989.

[26] W.A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.