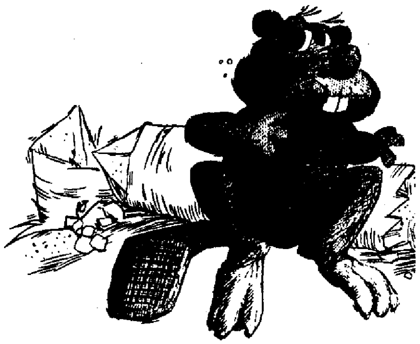
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT



Anthropomorphic Programming

Kellogg S. Booth
Jonathan Schaeffer
W. Morven Gentleman

CS-82-47

February, 1984

Anthropomorphic Programming *

Kellogg S. Booth
Jonathan Schaeffer

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
(519) 886-1351

W. Morven Gentleman

Division of Electrical Engineering
National Research Council of Canada
Ottawa, Ontario, Canada K1A 0R8
(613) 993-2629

Anthropomorphism is the attribution of human form or personality to non-human objects. Ancient Greeks anthropomorphised the gods in order to more easily comprehend their actions. Modern programmers may find that anthropomorphism provides similar insight into the behavior of parallel programs. Anthropomorphic programming is a proven technique for building systems that work using multitask structuring and message passing schemes to simplify the analysis of complex systems.

These principles are illustrated by examples from the literature and from ongoing research in which multitask structuring results in highly modular, portable systems. Many common problems associated with large systems (deadlock prevention, performance monitoring, load balancing, resource allocation, debugging, and synchronization of activities) are vastly simplified when they are treated anthropomorphically, often to the point that solutions mimicking structures evolved within human society can be applied immediately to problems in software engineering. Anthropomorphism is thus more than just a linguistic artifice. Its metaphors are powerful tools for system design.

* This research was supported in part by the Natural Sciences and Engineering Research Council of Canada under various grants.

Introduction

With the advent of parallel computers it is possible to structure programs as a collection of cooperating tasks that execute concurrently. In order not to be overwhelmed by the complexity of programming, it is important to have metaphors and idioms with which to organize the tasks. Anthropomorphic programming is data flow at the subroutine level, with the added feature that metaphors (anthropomorphisms) are used to "flesh out" the design and consequently our understanding of the program structure. This approach to design forces the programmer to explicitly address parallelism inherent in the problem. By relating this concurrency to similar cooperative tasks in everyday life, the programmer is frequently able to develop elegant yet simple solutions. Our interest here is not merely academic; elegance and simplicity contribute directly to our ability to comprehend a program's behavior.

The use of multiple tasks allows explicit scheduling of operations to be postponed as long as possible, or even suppressed entirely. This delayed binding accomplishes for the scheduling of operations what relocatable modules and segmentation schemes accomplish for the placement of data in memory. Multitask structuring is thus a mechanism for resource allocation in the temporal domain that parallels techniques for resource allocation in the spatial domain. Message passing is the means by which tasks communicate and the means by which synchronization is achieved. A principal advantage of this approach is that it can be made independent of linguistic features. Robust message passing primitives can be embedded in Fortran or any other programming language.

Our attitude about concurrency is counter to other well-promoted design methodologies for breaking monolithic sequential programs into parallel structures. We see a confusion concerning the role of explicit parallelism within computation. The source of this confusion is the bundling together of two very separate notions: the asynchronous nature of actions within a computation and the scheduling of those asynchronous actions. These are different issues and should be dealt with separately. Implementors have attempted to provide high-level languages that automatically detect parallelism during the optimization and code generation phases of a compiler, effectively allowing users to ignore both issues. We think this is a mistake. We advocate a different methodology for designing programs. Computations should be expressed asynchronously unless there is an explicit dependency compelling synchronization. Programmers should be forced to face the asynchronous nature of problems; the question of scheduling should be addressed separately, on its own merits.

The concepts of "task," "process" and "computation" have various definitions within the literature. The terminology we will use here will not be rigorously defined because we intend that our ideas apply to many of the concepts discussed by other authors. Rather, we will simply say that a task is roughly equivalent to our intuitive notion of a subroutine, but with the added distinction that when the task is created it executes concurrently with its creator, perhaps even in parallel. This contrasts with the simpler notion that a subroutine only executes when called and that the caller blocks pending completion of the subroutine. Implicit synchronization is lost when we allow concurrent execution of tasks. Most applications will thus require that tasks be able to communicate and synchronize; we will assume that a message passing mechanism exists for explicit synchronization [14]. The choice of message passing over other synchronization primitives is not central to our argument.

History

It is difficult to pinpoint the first occurrence of the term "anthropomorphic programming." The most general form is probably due to Carl Hewitt [16], [17], [18], [19], [27]. He interprets any intelligence, whether the ability to retrieve a stored value or the ability to perform an operation, as the role of an actor.

In Hewitt's terminology, each actor (task) follows a script (program). The script defines how the actor will respond to receiving any possible message. An actor is only defined in terms of its behaviour. A program is the pattern of interaction between the cast of actors, each following a script. Teams of actors normally are required to do anything interesting. Hewitt's work provides a theoretical basis on which others have built. His abstractions are embodied in many of today's attempts to structure a computation as a collection of tasks working together toward a common goal. The Smalltalk-80 system [15] is one of the best known examples. Capitalizing on Hewitt's ideas with available hardware requires that an intuitive understanding of his goals be tempered by practical considerations of efficiency.

Some earlier manifestations of anthropomorphism are found in the work of Edsger Dijkstra in which he discusses concurrent tasks [9], [10]. Here is how Dijkstra advises us to view a large programming system.

> "If we dare to regard the whole happening as 'meaningful,' we do so because we have mentally grouped sequences of instructions in such a way that we can distinguish a structure in the whole happening . . . the structure is *our* invention and *not* an inherent property of the equipment . . . it then follows that it is the programmer's obligation to structure 'what is happening where' in a useful way."

Dijkstra devotes over four pages to describing his technique as applied to the problem of the "Five Dining Philosophers." He presents a rather elegant solution using semaphores that synchronizes the philosophers' use of their shared forks. A generation of PhD students writing comprehensive exams will attest to the subtlety of his arguments. But toward the end of his discussion we encounter the following remarkable sentence.

> "Instead of N sequential processes cooperating in critical sections via common variables, we take out the critical sections and combine them into a[n] N+1st process, called a 'secretary;' the remaining N processes are called 'directors'."

Dijkstra goes on for another page or so to tell us how this scheme works. But we suspect that most people do not have to read beyond that initial sentence to grasp the complete solution. The inescapable conclusion is that the simplicity of Dijkstra's explanation, and the appeal to everyday experience, are all the proof we need. Curiously, seven years later [11], Dijkstra informs us that "The use of anthropomorphic terminology when dealing with computing systems is a symptom of professional immaturity." We obviously disagree.

Anthropomorphic Design

In his own words, Dijkstra has "used the metaphor of directors and a common secretary." His analogy with real-life leads to an immediate understanding of the situation. Two points are worth reiterating here. The first is that the solution itself is easy to comprehend once given. Equally important, however, is the observation that recognizing in the problem the need to coordinate the actions of a number of independent computations, we can draw from our experience similar situations in which society has evolved solutions. Armed with this insight, we can realize an advantage using anthropomorphic metaphors.

Our aim is to embody within a task the same level of intelligence associated with a human being in an organization. We do this in a number of ways, most easily by adopting anthropomorphic metaphors. The concept of programming metaphors is a basic tool for thinking and reasoning about programs. This is most pervasive at Xerox PARC, where the term "metaphor" has itself become a metaphorical way of speaking about program abstractions.

De Millo, Lipton and Perlis presented a paper at the 4th POPL Conference called "Social Processes and Proofs of Theorems and Programs" [8]. The authors mean the first two words of the title to refer to the informal mechanisms by which mathematical theorems come to be believed by the community of mathematicians. There is another interpretation, however, which might occur to those for whom the word "process" is a synonym for "task," in which the words "social processes" refer to anthropomorphised abstractions of programs. The remainder of the title then assumes a new meaning, different from that originally intended by the authors, but still (we believe) within the spirit of their arguments. Suitably anthropomorphised, programs become less fearsome (like the gods of the Greeks) and consequently easier to live with. Following our intuition, we come to believe the correctness (or detect incorrectness) of programs by appealing to our own social experience. It is easy to find examples that illustrate this phenomenon.

A solution to the problem of the Five Dining Philosophers has been proposed by Cargill [4]. He replaces the forks by a setting of alternating forks and spoons (as would be found at a real spaghetti feed) for which there is an easy protocol that insures all of the properties desired in a complete solution; each philosopher picks up his fork before his spoon. The point here is not that a simple solution exists, but rather that the explanation of the solution is so transparent once the problem is viewed in everyday terms.

It is worth noting in this regard that a solution similar to Cargill's has appeared, but without the anthropomorphism [3]. The correctness proof provided by Burns has at least two (patchable) holes. The program as written does not work. The solution is not "wrong," but it is not immediately usable. We claim that this is not an isolated phenomenon. Despite the many efforts in structured programming (or perhaps because of those efforts) it remains a difficult job to design and implement reliable programs. Any mechanism, however simple, which aids us in that endeavor should be given serious consideration. This is summed up for us by Dyment [12].

> "With all of this, however, the intellectual management of a large software project remains a matter of considerable difficulty. Well defined abstractions for dealing with the inherent difficulties associated with real-time interactions, multi-user configurations, parallel execution, and just the physical size of complex tasks often appear unavailable."

What Dyment calls for is a formalism that will aid in the intellectual management of software projects. Those with a mathematical background will be quick to point out that the tools of program verification offer such a formalism. We again cite De Millo, Lipton and Perlis.

> "We believe that, in the end, it is a social process that determines whether mathematicians feel confident about a theorem – and we believe that, because no comparable social process can take place among program verifiers, program verification is bound to fail. We can't see how it's going to be able to affect anyone's confidence about programs."

We have already pointed out that those authors meant "social process" to refer to mechanisms in use among the community of mathematicians, not to our notion of anthropomorphism. But our use of their argument is not out of place. An anthropomorphic view of a program as a team of processes can provide just the "social, informal, intuitive, organic, human process" that De Millo Lipton and Perlis find to be at at the heart of successful mathematics. We suggest that a proper goal of programming is to come to understand programs in the same way that we eventually come to understand people, even though we may never be entirely successful in predicting the behavior of either.

Case Study: Public Key Encryption

Needham and Schroeder [23] posed a problem in signature authentication whose solution required an implicit network-wide time standard to validate signatures. Their purpose was to provide a "foolproof" guarantee that digital signatures were in fact correct. The primary deficiencies of their scheme are its dependence both upon a network time standard and, worse, the assumption that encryption keys are never compromised. Dealing with real distributed systems necessarily makes the first assumption unlikely. The second assumption is dangerous in all cases. We previously put forth a different solution which has neither of these drawbacks, but which instead relies on the well-known function of a notary [2]. Our anthropomorphic approach uses a time-tested social institution to guarantee correctness.

With Needham and Schroeder's scheme, in order to sign a document such as a check the maker encrypts the document with his secret key. The payee relies upon the fact that the decryption capability of the maker's public key is tantamount to proof of authenticity because (in a well chosen public key system) the probability of malicious or accidental decryption is effectively zero. The problem not addressed by Needham and Schroeder is the question of compromise. Should the maker subsequently announce that his key has been "compromised," the payee is left with no guarantee. The system leaves much to be desired, since an unscrupulous maker can renounce his key whether or not it has actually been compromised.

As in real life, our proposed solution gives a user (server, task, actor) within the system the job of verifying signatures and validating their authenticity – not when that authenticity is contested, but instead when it is not contested, at the time of the signature. The notary, a neutral party, signs the document in a similar manner to the maker, but using the notary's secret key. Subsequently the payee has two means for proving authenticity. If the maker's key has not been renounced the original signature is obviously "good." If the key has been renounced, the signature of the notary can be relied upon as an assurance that, at the (unspecified time) that the document was notarized, the signature of the maker was indeed valid.

A convincing "proof" that the solution satisfies our claims is little more than an appeal to our everyday experience with signing documents and our social trust in notaries. It happens that anthropomorphism aided in our discovery of this solution, but we believe the strongest case for anthropomorphism is its explanatory role when dealing with algorithms for inherently complicated situations. The reader who doubts this need look no further than the original implementation of the Knuth-Pratt-Morris pattern matching algorithm to see the importance of being able to understand, at an intuitive level, the workings of a program. Originally installed within a text editor at UC Berkeley, the pattern matching algorithm was removed when a maintenance programmer realized that he could not understand how it functioned. It was replaced with a significantly slower, but known-to-be-reliable algorithm [28]. The maintenance programmer was correct in his decision.

Programs That Work

While it may be overstating our case to claim categorically that existing large organizations do work and existing large programs do not, it is certainly true that the level of asynchronous activity is far greater for large organizations than for large programs. The question is whether we can capitalize on this experience and turn it to our advantage. In this respect we may be victims of our own cleverness. Forty years ago "large" physics calculations were performed by groups of people using mechanical calculators. As sequential computers replaced manual team effort, coordination among subcomputations was neglected. Had advances in hardware been slower we might have been forced to address the problems of organizing large teams of human computors.

How were these people organized? Surely there must have been some coordination between individuals working on the same calculation. Examining the problems in such an organization could provide useful insight into how we should design programs for those same calculations. For example, in a social context it is essential that provision be made to limit the repercussions of poor performance or outright failure of individuals and to minimize the impact on other workers' activity during recovery. This is a notorious shortcoming of most programming systems. Failure of a single software module frequently dooms an entire project.

The goal in programming is to produce a program which solves a problem. First and foremost a program must run and it must perform in the way we intend. Efficiency is usually a secondary consideration. This attitude toward programming is fostered by anthropomorphism, which hides the non-essential complexities of a problem while highlighting the key aspects of a situation. The key aspect in this case is the abstraction of behavioral responses to communication among tasks. This focus allows the systems designer to separate the specification of services from their implementation.

The delayed temporal binding afforded by multitask structuring means that the programmer can complete the entire program without worrying about scheduling considerations. Only after the algorithms have been tested and are seen to deliver the desired result does the question of efficiency become important. As the program evolves it is easy to allow more concurrency. Our experience has been that, contrary to concern expressed in the literature, errors such as critical races seldom occur in real programs designed this way. If anything, the designer usually errs on the side of conservatism and over-constrains the parallelism.

A much more pervasive source of errors occurs at the level of specification. Statistics compiled from error logs show that almost as many programming bugs arise from incorrect specification as from incorrect implementation [13], [25]. Undoubtedly some of this can be attributed to a poor understanding of the original problem, but we think it more likely that the primary cause is the specification vehicle itself. Some efforts at solving this problem have attempted to increase the precision of the specification, thereby introducing a formal notation foreign to both the specifier and the implementor, guaranteeing that neither fully comprehends the implications of the design. In our world, anthropomorphic design would change the specification to be within universal experience, reducing the probability that the specification will be misunderstood, either by the specifier or by the implementor.

Case Study: An Interactive Paint Program

A good example of the way in which anthropomorphism can be used to simplify the conceptual design of a system is the interactive paint program implemented for raster graphics hardware at Waterloo [1], [24]. The paint program runs under the Thoth operating system, a message-based portable operating system [6]. The program is divided into a number of tasks, each assigned a specific job, much like the division of labor among people working together toward a common goal. Some of these tasks have subsequently migrated into microcode and could well evolve into hardware in future versions. Multitask structuring provides a means for ensuring that this capability is maintained at a high-level within the system.

Unlike more traditional implementations of painting systems, we have been able to incorporate algorithms directly from the literature, rather than having to elaborate on them in order to handle asynchronous events such as user-initiated breaks. A specific example of this is the algorithm which fills an area on the screen given a seed point within the area. It frequently occurs that the area is not a closed region, in which case the paint program "spills" into adjacent areas. When this happens, the user must be able to signal the program that filling should immediately cease. This is accomplished in our implementation by having separate tasks, one to perform the actual fill, a second to wait for a possible signal from the user, and a third to oversee the entire operation. The result is that each task consists of a very simple piece of code to perform a specific job, in contrast to a larger hodge-podge of instructions that attempts to interlace the two activities of filling and detecting signals from the user.

Similar constructs are used throughout the paint program. The anthropomorphic roles of administrator, secretary, overseer, agent and assassin have emerged as useful tools for structuring programs with the message-passing primitives available in Thoth exploited to provide synchronization.

The Mega-Buck Physicist and His Problems

This paper grew out of a talk we originally intended to present at a conference sponsored by the physics and computation groups at two government laboratories. We were planning to discuss issues of anthropomorphic programming in the context of the multitask paint program, but were persuaded that this was not what we really wanted to do. The comments below are George Michael's [22], after he read the abstract we sent to the conference organizers.

> "I do have a reaction to the abstract. In the following I'm wearing a mega-buck physicist's hat; 'It is rather impressive that they have designed an anthropomorphic interactive paint program. It is cleverly able to run on home computers under control of some Egyptian god! Wow!' The mega-buck physicist has a different problem. He doesn't know any better mathematics so he uses large computers to do large computations. He needs to do more because the models are weak, so he is looking for parallel and vector processors. He does not see general ways to divide the computation so cleanly as you can in the paint program. He is not likely to listen to you. Your approach is fine for separable tasks but not for problems at LASL and LLL."

George has a valid criticism. One which needs to be addressed. We grant that many physics codes do not seem to exhibit much separability, at least when we first look at them. Nevertheless, large codes *always* have asynchronism. The user's interaction with a program is the most obvious example. This includes not only input and output, but also debugging and performance monitoring. The complex nature of large physics problems invariably makes graphical display of intermediate and final results essential. Again, this interaction with the main code is fundamentally asynchronous. Moreover, the user's viewing of the data obtained through this interaction is also an asynchronous activity. A user may want to interactively peruse one or more models of a problem to compare results. The depth of exploration depends heavily upon what the user sees in the various models.

Although large codes traditionally have been written as monolithic synchronous programs, there are advantages to expressing parts of them as separate tasks. Some large codes analyze aggregates; transient stability analysis of power grids is a case in point. Various generators and loads are modelled separately and can profitably be assigned to individual tasks. Even if this is not the case, there is still a use for parallelism. A good example is the familiar problem of overlapping input/output functions with the main computation. Multiprogramming systems solve this problem quite handily for the general user, but in a large physics code where resources are already inadequate, the possibility of achieving overlap through sharing with other users is not feasible. The large code must somehow share cycles with itself. This is usually accomplished using a machine-specific intertwining of I/O and computation to efficiently schedule operations. If the original expression of the computation were in terms of multiple tasks the computation could multiprogram with itself. Explicit scheduling might no longer be necessary.

One effect of this partitioning is that it may be possible to contain the ill-effects of failure within a single task so that the computation in other tasks can be salvaged. Clearly the partitioning allows exploitation of multiple processors. In both cases multitask structuring delays the binding to a specific scheduling, rather than forcing the decision at the start of the design cycle.

Numerous studies have shown that the kernel in large codes accounts for only a small fraction of the total lines of source. Beyond the numeric representations of the physics there are such matters as input and output, updating and manipulating the data base, generating output reports, and selection control and options. The kernel is even a smaller percentage of the total programming activity when this support software is considered. All of the code must be organized. Anthropomorphism suggests natural ways to achieve this structuring and provides greater assurance that the organization will carry over to other problems and to other hardware.

Linguistic Considerations

Most of the examples we have cited involve systems written in languages other than Fortran. Some languages contain message passing intrinsically. If we restrict the activity of actors to something on the order of a subroutine, little advantage is gained through such specialized notation. Subroutine calls can acceptably implement all of the necessary mechanisms. This is as true in Fortran as in other languages. It has the advantage of avoiding rigid semantics, allowing flexibility in the choice of message passing schemes. In contrast, Ada enforces a particular semantics which, we observe, does not correspond to any of the message passing systems with which there is practical experience.

We personally prefer BCPL-based languages, but realize the need to stay within the Fortran tradition for many applications. Our approach allows a free choice of programming language because the message passing primitives can interface to most languages through the standard subroutine or procedure facility.

There is a serious question as to whether other synchronization primitives such as monitors can ever offer this same degree of linguistic independence. This particular problem is the least of our objections to monitors, but it suffices to dismiss them from consideration.

Anthropomorphic design naturally leads away from what Steve Johnson has called the "center of the world syndrome" [20]. Most programming languages, especially Fortran, assume a master/slave relationship between the various components of a computation. But programs rarely exist in isolation. Usually, if a program performs well and if the program solves a real problem, the desire to use it in concert with other programs will arise. This frequently is very difficult to do because most programs are written as if they occupy the central role in the computation. Hewitt might say that the problem is one of having to continually revise the script. Actors don't want to play bit parts, each wants top billing. Programs must be capable of playing supporting roles as well as the lead. This gives an entirely new interpretation to Weinberg's idea of "egoless programming" [26].

Dynamic Behavior Monitoring

One activity which is always asynchronous with the main stream of the computation is monitoring the dynamic behavior of a program. This activity is a generalization of the way that interactive debuggers are normally used. Even in a situation of provably correct programs it can be important to interactively monitor various internal structures to fully understand performance issues. Often this involves deriving and displaying ancillary results based upon available intermediate values.

Two special cases of behavior monitoring are of particular importance. Program debugging is a familiar feature in most systems. Robust exception handling is less often found. Invoking the analogy with social institutions, frequently the most effective means of exception handling is to "go over the head" of the person (task) committing the error. In anthropomorphic systems this is easy. Special tasks can readily be incorporated into the system which act as "spys" or "voyeurs" to detect aberrant behavior and to report to other tasks ("supervisors") for appropriate action. One way to view both debugging and exception handling is that one part of the computation is monitoring another, its action dependent upon finding some particular pattern in the computation flow.

Case Study: A Text Editor

Our final example is again drawn from the Thoth system. The text editor in Thoth appears to the user much like many other Qed-based editors [5], [7]. It is line-oriented with a minimal regular expression capability. What is strikingly different is its internal structure. The editor consists of two independent tasks. One, the actual "master," is the buffer proprietor. This task accepts messages from a second editor task. The messages consist of instructions to modify text in the file being edited.

The purpose for this division of labor is twofold. First, each of the tasks is much easier to conceptualize. It has only one job to accomplish, not two. Second, error recovery is particularly simple. After each atomic change to the file, the buffer proprietor awaits further instruction from the editor task. Should the editor task become entangled due to an attempt to perform an illicit act, it simply commits suicide rather than unwind its potentially complicated internal state. Should the user decide to abort an action, the break key automatically destroys the editor task. In either case the buffer proprietor detects the death of the editor task and creates a fresh instance of the editor which begins anew. Because the actions of the buffer proprietor are small and atomic there is no possibility of its failing. The isolation of the buffer proprietor guarantees the integrity of the text file throughout the editing session.

Current Work

The examples cited above illustrate various applications in which anthropomorphic multitask concurrency aids in the design, implementation and understanding of programs. Much of that work was performed in the Thoth environment at the University of Waterloo. Development on Thoth ceased about two years ago, but the notions of multiple tasks communicating through messages has continued in the Waterloo Port system, designed and implemented by the Software Portability Laboratory at the University of Waterloo, and in the Harmony operating system developed at the National Research Council of Canada.

A version of the multiple task Paint system has been implemented under Waterloo Port on an IBM PC. The work on Harmony is supporting research into intelligent, sensor-based robotics. Thoth was a uniprocessor system. Both Port and Harmony have extended the notion of tasks and message passing to multiple processors, Port in a loosely-coupled environment of personal workstations and Harmony in a tightly-coupled high-performance environment.

The acceptance of parallel computation is becoming widespread. Less common is an understanding of methodologies for supporting parallel computation within programs. Our advocacy of multiple task concurrency with synchronization via message passing is one such methodology. Karp [21] has pointed out that experience with MIMD architectures invites an analogy with human organizations. For a very small number of processors (people) detailed interactions can be maintained without a manager; with a modest number of processors (people) the interaction patterns can be handled by simple structuring techniques that decompose the problem into distinct tasks with well-defined areas of responsibility; with a very large number of processors (people) the interaction becomes so complicated that more rigid organization imposing a high degree of regularity seems to be required. We suggest that these three levels of complexity may be understood best through the metaphors of anthropomorphic programming.

Acknowledgements

Our colleagues have contributed much to our understanding of the ideas expressed here. A number of the examples we have cited are drawn from research conducted by members of the Software Portability Group including Michael Malcolm, Burt Bonkowski, Tom Cargill, David Cheriton, Morven Gentleman, Gary Sager and Gary Stafford. Many other people participated in the evolution of these techniques as they apply to problems in various areas such as computer graphics, numerical computation and computerized problem solving.

The present version of this paper is the result of encouragement by George Michael. A preliminary version was presented at the Conference on Language Issues for Large-Scale Computing held March 16-18, 1982, at Gleneden Beach, Oregon. The conference was sponsored by the U.S. Department of Energy, Los

Alamos National Laboratory and Lawrence Livermore National Laboratory.

References

[1] Richard J. Beach, John C. Beatty, Kellogg S. Booth, Eugene L. Fiume and Darlene A. Plebon, The message is the medium: multiprocess structuring of an interactive paint program, *Computer Graphics 16:3* (July 1982) pp. 277-287.

[2] Kellogg S. Booth, Authentication of signatures using public key encryption, *CACM 24:11* (November 1981) pp. 772-774.

[3] James E. Burns, *Complexity of Communication Among Asynchronous Parallel Processes*, PhD thesis, Georgia Institute of Technology (January 1981) Technical Report GIT-ICS-81/01.

[4] Thomas A. Cargill, A robust distributed solution to the dining philosophers problem, *Software – Practice and Experience 12:10* (October 1982) pp. 965-969.

[5] David R. Cheriton, *Multi-process Structuring and the Thoth Operating System*, PhD thesis, University of Waterloo (1979) Technical Report CS-79-19.

[6] David R. Cheriton, Michael A. Malcolm, Lawrence S. Melen, and Gary R. Sager, Thoth, a portable real-time operating system, *CACM 22:2* (February 1979) pp. 105-115.

[7] David R. Cheriton, *The Thoth System: Multi-Process Structuring and Portability*, North-Holland (1982).

[8] Richard A. De Millo, Richard J. Lipton and Alan J. Perlis, Social processes and proofs of theorems and programs, *CACM 22:5* (May 1979) pp. 271-280.

[9] Edsger W. Dijkstra, Co-operating sequential processes, in *Programming Languages*, F. Gunuys (Ed.), Academic Press, New York (1968) pp. 43-112.

[10] Edsger W. Dijkstra, Hierarchical ordering of sequential processes, in *Operating Systems Techniques*, C. A. R. Hoare and Perrott (Eds.), Academic Press (1972) pp. 72-93.

[11] Edsger W. Dijkstra, EWD498: How do we tell truths that might hurt?, reprinted in *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag (1982) pp. 129-130.

[12] Doug Dyment, A corkscrew for the software bottleneck, *Micros 1:2* (October 1980) pp. 21-24.

[13] Albert Endres, An analysis of errors and their causes in system programs, *IEEE Transactions on Software Engineering SE-1:2* (June 1975) pp. 140-149.

[14] W. Morven Gentleman, Message passing between sequential processes: the

reply primitive and administrator concept, *Software – Practice and Experience 11* (1981) pp. 435-466.

[15] Adele Goldberg and D. H. H. Engalls, The Smalltalk-80 system, *BYTE 6:8* (August 1981) pp. 36-48.

[16] Carl Hewitt, Peter Bishop and Richard Steiger, A universal actor formalism for artificial intelligence, *Third International Joint Conference on Artificial Intelligence,* Stanford University (1973) pp. 235-245.

[17] Carl Hewitt, Viewing control structures as patterns of passing messages, *Artificial Intelligence Journal 8* (1977) pp. 323-364.

[18] Carl Hewitt and Henry Baker, Laws for communicating parallel processes, *IFIP Congress Proceedings* (1977) pp. 987-992.

[19] Carl Hewitt and Henry Baker, Actors and continuous functionals in *Formal Description of Programming Concepts,* E. J. Neuhold (Ed.), North Holland (1978) pp. 367-387.

[20] Stephen C. Johnson, personal communication.

[21] Alan Karp, pérsonal communication.

[22] George A. Michael, personal communication.

[23] Roger M. Needham and Michael D. Schroeder, Using encryption for authentication in large networks of computers, *CACM 21:12* (December 1978) pp. 993-998.

[24] Darlene A. Plebon and Kellogg S. Booth, Interactive picture creation systems, Technical Report CS-82-46, University of Waterloo, Waterloo, Ontario, Canada (December 1982).

[25] Raymond J. Rubey, Joseph A. Dana, and Peter W. Biche, Quantitative aspects of software validation, *IEEE Transactions on Software Engineering SE-1:2* (June 1975) pp. 150-155.

[26] Gerald Weinberg, *The Psychology of Computer Programming,* Van Nostrand Reinhold (1971).

[27] Akinori Yonezawa and Carl Hewitt, Modelling distributed systems, *Fifth International Joint Conference on Artificial Intelligence,* Massachusetts Institute of Technology (1977) pp. 370-376.

[28] (unknown), possibly apocryphal story once heard by the authors and thought to be true by at least two other people who we asked.