A Compiler for Composition: CHAIMS

Louis Perrochon, Gio Wiederhold, Ron Burback
Computer Science Department
Stanford University, Stanford CA 94305

Abstract

CHAIMS supports an innovative paradigm in software engineering: Composition. The CHAIMS programming language focuses solely on integrating so-called megamodules into new applications. In doing this, CHAIMS exploits existing or emerging standards for interoperation like CORBA, ActiveX, JavaBeans or DCE. This approach reduces software development and maintenance cost by actively supporting autonomy and reuse of megamodules.

1 Introduction

There is a shift in the software generation paradigm from an emphasis on programming as code generation to an emphasis on composition and integration. In large-scale applications the modules to be invoked are typically distributed over many nodes on high- or mixed-performance communication networks [1]. Today, there are a variety of interface standards to support module interoperation and client-server and mediated architectures, all competing for primacy in the market. Just as once the competition was on 32-bit vs. 36-bit vs. 60-bit hardware, competition is now among module interfaces, client-server and object-oriented conventions and standards [2]. Rather than waiting for the final comprehensive interoperation standard, the application provider should be independent of any specific standard. To achieve this objective we propose a high-level language to overcome interface and network architecture differences and gain independence for large-scale applications. Hardware independence was achieved first by programming languages as FORTRAN IV, Lisp 1.5, COBOL, and Ada. However, integration support today is still platform dependent, even when languages used to program the software are not [3]. Reuse, essential to overcome the huge cost of software, is also hindered by the variety of interoperation standards [4].

The CHAIMS *(Compiling High-level Access Interfaces for Multi-site Software)* project, is developing a unique tool for software composition. The intended result is a very high level programming language to be used exclusively for software module composition. Its compiler will generate a variety of invocation sequences for software interoperation. These sequences are compatible with the standards becoming available for client-server architectures; CORBA [5], OpenDoc [6], ActiveX (based on OLE/COM [7]), JavaBeans [8], DCE [9], DIS [10], DOE [11], ILU [12], etc. As of now, we are just starting programming hard-wired prototypes in CORBA, ActiveX, and DCE.to experimentally validate the concepts.

2 Objective

Managing large-scale software remains a task which requires many levels of expertise, well-defined processes, adherence to standards, and careful documentation. Even when all these pre-requisites are in place, overruns and failures are common. We describe an experimental approach which attempts to reverse this process. Rather than following the waterfall model or its variations by starting from a design, CHAIMS assumes that large programs can be composed from existing modules. This limits the application programs to the composition of available resources. To broaden the range of resources, however, a CHAIMS megaprogram can access modules using any of a variety of existing interoperation standards, exploiting ongoing work in client-server technology.

CHAIMS hence focuses on interoperability with multiple interoperation standards, invoking modules on multiple computers, written in multiple source languages. Fundamental is a widely distributed and asynchronous operation. Later versions are planned to support optimization of dataflow among the distributed modules. The language includes the ability to set up module interfaces prior to executions, request performance estimates from modules prior to their invocation, schedule module execution in parallel, monitor execution of invoked modules, interrupt inadequately performing modules, and provide data and meta-information to customer interface modules [13].

CHAIMS supports the paradigm shift which is already occurring in building systems: a move from the focus of *programming* to a focus on *composition*. This shift is occurring invisibly to many enterprises, since there is no clear boundary in moving from subroutine usage to remote service invocation. But there are few tools and inadequate guidelines to deal with this change. Ten years ago, integration of large-scale software was performed by experienced groups in large companies, as in IBM, Unisys, Fujitsu, Arthur Andersen and the other `Big Five', and system contractors as SAIC, Lockheed, MITRE, Lincoln Labs, etc. Today, software composition has moved to UNIX and PC platforms, and an increasing fraction of the software workforce is engaged in composition, programming in the large with a decreased need for programming in the small [14 p. 201] [15 p. 8]. CHAIMS intends to fill the gap in composition tools.

Many early general purpose programming languages have had some composition facilities added to their basic capabilities. Examples are the LEAP feature in SAIL [16], modules in PLITS [17], the Courier Protocol in Interlisp [18 p. 21.7], rendezvous in Ada [19, 20], and tasks in PL/1 [21]. In the end, these facilities were unwieldy and did not enter common usage. Some information systems have had dedicated languages that controlled distribution, as databases (SQL) [22], report generators, and distributed information systems, with the intent to overcome the complexity of serving both programming and composition [23]. The Microsoft environment has had success with Visual Basic [24].

The CHAIMS megaprogramming language serves only module composition and scheduling. Its narrow focus should allow it to remain simple, although some significant new concepts are introduced. The size of the modules we envision typically justifies a dedicated processor, although modules can share a single processor when performance demands are modest. Modules written in C, C++, Ada, FORTRAN, etc., will need interfaces (similar to APIs) to allow interoperation in the CHAIMS setting. Interoperation protocols from standards as CORBA, ActiveX, JavaBeans and DCE provide, in effect, our machine languages and make the CHAIMS concept implementable today. We believe we can succeed because we can build on these efforts expended on interfaces for interoperation within client-server models [25]. We do not expect to develop new interfacing standards, but instead demonstrate the utility and the weaknesses of existing standards, since they are the primitives for our megaprogramming language. At the same time, by moving up one level of abstraction in programming, we are moving to a new paradigm where the focus is on composition of services, rather than assembly of code .

The CHAIMS project has limited scope: CHAIMS supports only a few data types, and the CHAIMS environment uses many defaults rather than give the programmer a rich palette of choices in the invocation of software modules. Also there is no plan for automatic programming. Problems of argument and logic composition have troubled mathematics for many years. CHAIMS has little dependence on the mathematical underpinnings because CHAIMS is a programming language, and not an automatic code generation or composition system.

By focusing CHAIMS on interoperation in a multi-site environment, rather than on platform-specific code, we gain high-level support of megaprogramming concepts:

1. Flexibility of interface standard choice, changeable at any time by megaprogram recompilation.
2. Composition clarity by presenting executable instances of the domain-model architectures.
3. High system performance, since asynchronous and parallel operation is the default.
4. Life time maintenance as a seamless continuation of system development.
5. Adaptability of applications to evolving interface and product standards.

   In a follow-on phase we expect also to address

6. Optimization between autonomous megamodules.

Each of these goals is elaborated below in Section 4. It must be realized that module performance, although not addressed directly by CHAIMS, continues to critically affect system performance. In a multi-site system composed via a CHAIMS megaprogram, individual module performance will be improved by allowing a choice of platforms and module programming languages. [13] discuss in detail further megaprogramming concepts like the possibility for testing of new and updated software without committing to specific configurations, adaptation to change of scale of modules, or network configuration.

3 Related Work

Not surprisingly, there are many other initiatives which address the same problems that motivate CHAIMS. Fortunately, they tend to be complementary rather than competitive. The need for component technology was already expressed in the 1968 NATO conference [26]. The early focus was on software for single processor nodes. Initial progress was slow, due to a lack of adequate programming concepts, until the object-oriented programming paradigm was established and supported by an acceptable and maintained language tool (C++) [27]. Today many object libraries are available, and they provide a basis for high productivity in writing single-processor programs [28]. Experience gained in industry with distributed objects helps in implementing CHAIMS.

Transitioning of object assembly to multi-site computing has often been discussed, but solutions have focused on single protocols. We profit conceptually from the experience gained in such efforts and profit practically from the interoperation protocols that have been developed. While it is accepted that distributed systems will be common, and we have some impressive handcrafted examples, such as the SIMNET and its successors [10], the tools needed to rapidly assemble, restructure, and maintain such systems are lacking. Now, useful infrastructure standards have been developed. For distributed simulation the DIS protocol provides the basis [29]. Similarly for distributed object data configurations the Interface Definition Language (IDL) provides the definition capability [5].

Initiatives which are relevant to CHAIMS include:

- Object Libraries: Collections of object class definitions that share a common domain ontology [28]. For instance there are object libraries for geometry, for planetary trajectories, for accounting, etc. These are typically assembled using a programming language. Except for remote Graphical User Interfaces (GUIs), most object libraries do not consider distribution, and the GUIs have very limited models of inter-computer interaction.
- Automation of Composition: When object libraries are coherent, automatic object composition according to a well-defined architecture becomes feasible [30]. CHAIMS does not go that far, but relies on the megaprogrammer to understand module semantics and resolve inconsistencies. Resolution may require the generation of new modules. Success with CHAIMS may encourage more composable modules to come on the market and even new initiatives in automation.
- Reuse of architectural knowledge: The Protégé approach established by [31] has permitted an impressive degree of conceptual reuse among quite different domains.
- Architecture Description Languages (ADLs): ADLs provide a conceptual and general definition of software architecture types, often in graphical form [32]. A CHAIMS megaprogram in effect describes an instance of the architecture of the application system. Since CHAIMS is to be executed, it is always solidly ground in existing resources while an ADL description can remain independent. The level of abstraction in CHAIMS is necessarily less than that of ADLs designed for abstract descriptions and the generation of descriptive graphics.
- Interoperation Standards: Real or potential standards for interoperation, as CORBA, ActiveX, JavaBeans and DCE become the machine-language for CHAIMS. We depend critically on them, and hope they will succeed in making client-server middleware an effective development concept. Their perceptual weakness is that they operate at a low level. Furthermore, the competitive nature of their promoters does not favor general interoperation. Their focus on serving clients in star configurations does not support optimization.

By isolating the concepts related to technology of component from these and other large-scale system initiatives, and thus reducing our conceptual scope, we expect to be able to make progress in module composition and reuse within a research project of modest size.

4 CHAIMS Architecture

The components of CHAIMS are symbolically depicted in Fig. 1. The topmost component is the (mega)programmer, who selects the modules to be invoked. In Fig. 1, three module have been selected. The megaprogram, written by the megaprogrammer, only manages the invocation of the modules, according to precedence constraints that typically represent dataflow dependencies. The modules themselves may use a variety of interface standards. Any needed data type or computational conversions to be performed in the dataflow are invoked externally to the megaprogram, assuring that the megaprogram is a clean representation of the architectural intent for this application. The megaprogram is compiled by the CHAIMS compiler. The resulting executable runtime program consists mainly of module invocations, expressed using the interface standard protocols that are appropriate for this module. Many of these invocations will involve remote accesses, since we assume that the modules reside on multiple sites, connected by a communication network. The actual modules are either pre-existing or written to order in a suitable programming language. The autonomy of the modules encourages reuse [33]

Since CHAIMS does not provide for automatic programming or knowledge-based techniques [34] the megaprogrammer must know which modules offer what

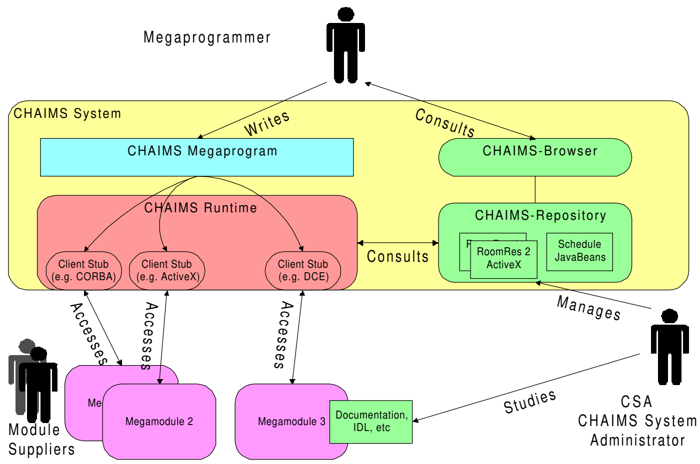

Figure 1: Components of the CHAIMS Megaprogramming Paradigm.

functionality and must have access to a well-maintained CHAIMS repository. The process of CHAIMS megaprogram generation and use is illustrated in Fig. 2. The megaprogrammer builds the source code of the megaprogram in the composition environment. During that process the megaprogrammer will access module descriptions in the repository and also obtain initial feedback from the CHAIMS compiler.

CHAIMS does not provide inout-output interfaces. The result of a megaprogram execution are simply shipped to an output module, written in any language that is compatible with the end-user or customer platform.

The megamodules are independent, but must support general megaprogramming concepts and services. This allows independent software vendors or content providers to develop new megamodules, that are easily accessed by CHAIMS programs. Modules composed by a CHAIMS program can be written in the most suitable language and use the best representation for the given sub-task and solution approach. However, especially in the beginning, legacy software can be wrapped to satisfy the new requirements [35, 36]. Where required, glue-code to resolve semantic differences will be needed to generates a compatible interface [37, 38]. Since services as DCE marshaling perform well in this task, the CHAIMS compiler itself will not be burdened with having to generate such glue code [9]. Neither will the CHAIMS compiler be burdened with code for fancy end user input/output. Interaction with the customer running the megaprogram dealt with through remote or local input/output megamodules.

5 Support of General Megaprogramming Concepts

There are two aspects of megaprogramming, process and tools: *Process management* has become a primary concern in recent years, leading to insights and guidelines for managing the sequence of steps leading from creation of reliable software to the reuse of existing software [39]. Reusing existing modules takes advantage of the validation conferred by practice [40]. Process concepts are largely valid independent of platform, hardware distribution and computer language selection, although suppliers will, of course, stress the relative benefits of their wares, as CASE tools, GUIs, and the like. However, this decoupling is also a weakness. Without tools the programmer has little reinforcement to adhere to the desired practice. The completion of certification forms is hardly a guarantee of optimal process, and is quite costly. Much paper is generated, but rarely revisited when changes or maintenance occurs. In DoD documentation costs amount to 50-60% of SW generation costs [41] [42, p.21].

The other aspect of megaprogramming, tools to achieve rapid *composition,* is a largely ignored area, except for meta tools [43] include graphic tools to describe system architectures, domain-specific software models, module repositories, languages for rapid prototyping, etc. While all these meta tools help, they do not provide the essential service to manage large-scale, distributed software as it must evolve over time [44]. CHAIMS focuses on the tools aspect. Having general, productivity-enhancing tools simplifies adherence to the megaprogramming process . Tools also provide an op-

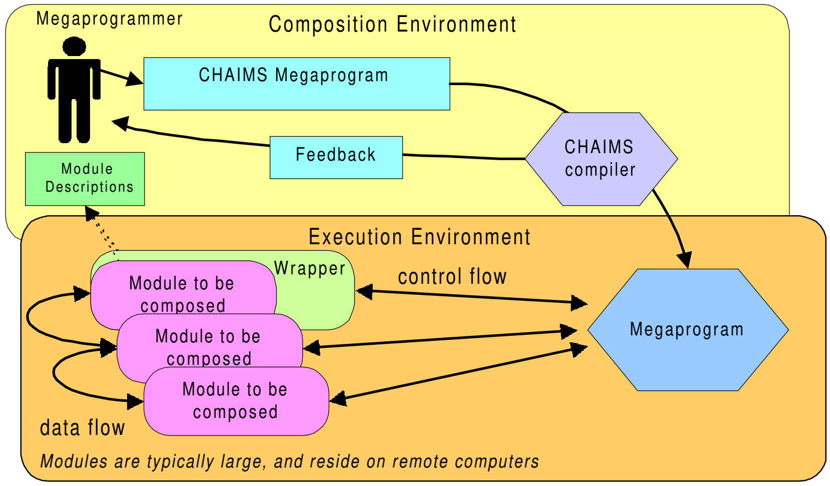

Figure 2: The CHAIMS Megaprogramming Process and Information Flow.

portunity to measure adherence directly, whereas today adherence to the process is measured by collecting documents and signoffs from the programmers and their managers [45]. Furthermore, the CHAIMS program, and any assessment tools bound to it, , will stay with the system beyond delivery, since it is the actual embodiment of the executing code, and not just a throw-away specification. Effective maintenance is economically crucial as 60-90% of system costs are incurred during the maintenance phase [46 p. 27, 47, 48 p. 69]

Flexibility prior to actual deployment: In CHAIMS, the binding of megaprograms to megamodules can be delayed arbitrarily, and changed later by recompilation of the CHAIMS program. This delayed binding relaxes commitments to specific interface standards. Choices among standard interfaces as CORBA, ActiveX, JavaBeans and DCE can be deferred until the scale of the problem being addressed is known and the platform capabilities can be assessed. Even new interface standards can easily be adapted by updating the compiler, its library and recompilation of existing CHAIMS megaprograms. This flexibility goes beyond the facilities that programming languages can provide in terms of platform allocation [49].

Composition clarity: The structure of a megaprogram defines the architecture of a system, independent of its implementation. In effect, the model architecture will be clearly visible, and is not buried in interface and application-specific code [50]. As such, it supports the concept of domain-specific software architectures (DSSA) [51]. Clarity of architecture, and autonomy of module development attacks the major problems leading to system failures [52].

The megaprogram defines an architectural instance of an application, while delegating all computational activities to the modules it invokes. It is, in effect, a model of a DSSA. The architecture instance defined in a megaprogram is reusable, not as a paper document, but when linked and recompiled with appropriate modules, as a complete re-instantiation of the domain architecture. Since the compilation can bind to alternate interface specification languages, scalability and platform independence can be achieved, as long as equally competent modules can be acquired. The megaprogram will be limited by the capabilities of current interface languages, but these are improving rapidly.

We do not intend to provide a graphic editor to manage a graphic description of the megaprogram's architecture. Many such tools are available, e.g., [53]. If we reach a state that the megaprogram becomes so large that it is hard to understand, such tools can be adapted. We hope that the megaprogramming approach reduces complexity so that such tools, are not needed [54]. Validation of a CHAIMS architecture instance is through its execution, including the execution of the megaprogram in smaller environments.

Parallel computation and asynchronous communication: Modeling the activity of real world objects, parallel operation of the megamodules is the underlying assumption in CHAIMS. This vision means that the increasing availability of distributed computing can be exploited without resorting to parallel computing features applied to sequential programming languages. By considering any sequential dependency as an exceptional constraint, the megaprogrammer will naturally think of parallel execution. This concept of natural parallelism is the alternative to the common paradigm in force today, where the programmer codes the actions to be performed in a world where everything lives in parallel into a sequential format, which is subsequently analyzed by parallelization tools to extract possibilities for parallel execution. Since action in a natural, distributed world actually occur in parallel, we hope that CHAIMS megaprograms can capture their essential parallelism without dual translations. CHAIMS should make it easier to train domain specialists in megaprogramming rather than untrain conventional programmers that are used to all forms of restrictions.

Maintenance as the economic driver: Inadequate investment has been cited by promoters of the reuse process as the reason for poor acceptance of their process models [33]. When the focus is on development, the case for investment in reusable modules is indeed hard.

CHAIMS addresses the development and ongoing maintenance of large, multi-site software systems. As indicated above, maintenance is the largest cost factor (60-80%) in modern software, and at the same time, often ignored in proposals intended to improve software engineering [55]. Since the megaprogram is an actual part of an application, it cannot be lost or become obsolete with respect to the executing code. Since it also represents the instance of the architecture description, this crucial information is retained at high level. Over the lifetime many modules will have to be replaced, but the architectural specification can remain invariant, unless module capabilities change semantically.

Adaptability to changing standards: The move to composed software is clear, but still poorly focused. We have as many proposed standards for software interoperation as we had computer hardware architectures thirty years ago. The instruction codes that defined a family of computers remained fairly constant over its lifetime, so that the code generation portion of code compilers needed little maintenance, once it was comprehensive and fault-free. Today dealing with hardware platforms is simpler: first, there are fewer choices, and second, most computer programming languages can deal with a wide variety of platforms, so that programmers are no longer tightly constrained by platform availability. There are still differences among classes of computers that must be recognized. Traditional personal computers process a single customer thread at a time, medium-scale computers are effective for multiple, cooperating customers, while main-frames support, large, heterogeneous customers. In each class, parallel computers are being introduced or distributed configurations are assembled to permit parallel execution [56].

Traditional programming languages have had as a major benefit platform independence, since most programs can be recompiled for new hardware. Application software typically lives about 15 years, much longer than its hardware. The same stability does not exist yet for interface software for large systems, which may involve multiple computers of differing classes.

A unique aspect of distributed programming is that it depends on interface standards that themselves are undergoing rapid evolution [2]. The interfaces for distributed computing are likely to stay fluid. For instance, SQL is undergoing changes in its transition to SQL-2 and -3. Selection and implementations of feature sets will vary widely. OMG's CORBA 2 will have many features not available in CORBA. KQML concepts are being suggested for future versions of CORBA [57], new standards with good chances for success arise like JavaBeans and gain attention within a few months. Just as traditional programming languages provide an insulation from platform differences, the essence of CHAIMS is to provided insulation for the megaprogrammer from the differences in today's interoperation architectures and standards. Such an independence is especially crucial for the larger systems which need to operate on multiple sites, utilizing networks and a variety of modules and services. These systems represent major investments, and have a long lifetime.

6 Optimization Between Autonomous Megamodules

Interoperation standards provide the infrastructure for the megaprogramming language CHAIMS. The function of a CHAIMS program is to set-up, invoke, and extract results from pre-existing or written-to-order modules, then execute them in a correct and efficient manner. Execution of multiple modules and the megaprogram itself occurs in parallel, unless prohibited by explicit constraints.The binding of standards selection of interfaces can be delayed up to execution time, although compiled CHAIMS megaprograms may have to be recompiled to adapt to new interface standards.

To achieve new flexibility in module scheduling, needed for optimization, we will introduce new module interface management capabilities. These will break up the functions of the traditional `CALL` statement, in the context of Remote Procedure Calls (RPCs) . A traditional `CALL` to an *application function* is replaced by several *base functions* to `SETUP` the application function with basic parameters, `INVOKE` it once or several times with modified arguments, `EXTRACT` the results etc. For optimization the `ESTIMATE` statement[13] reports the estimated execution cost for a megamodule. This removes the programmer's need to break module encapsulation in order to get the information needed for programmed module scheduling to gain adequate performance. Communication bandwidth requirements will also be reduced.

7 Status of CHAIMS

The CHAIMS project started in September 1996. We are currently analyzing details of CORBA, OLE, JavaBeans and DCE. Sample executable codes are being written to assess and compare the required invocation sequences. We are developing a hand-coded prototype of a CHAIMS runtime environment that can arrange meetings, make room reservations and can order a lunch with megamodules accessed using CORBA, OLE, and DCE. The progress of CHAIMS is fully documented on

`http://www-db.stanford.edu/CHAIMS/`

The ongoing, manual compilation process should give us enough experience to define an adequate syntax and semantics for the CHAIMS programming language. While CHAIMS will be a procedural language, its flavor may range from Ada to ML [58]. We hope that by the time the symposium convenes that this aspect will be clarified. It may be important to have the CHAIMS language not be similar to an existing sequential programming language, so that it will be clear to the megaprogrammer that the environment is inherently parallel, and that much prior experience and training will have to be unlearned.

8 Summary

The focus of the proposed megaprogramming language CHAIMS and its interface drivers is to reduce the cost of long-term maintenance and software evolution, and reduce the numbers of errors occurring in this process, without reducing the amount of maintenance and evolution actually being performed. The CHAIMS megaprogramming language only serves module composition and scheduling and will remain simple, although it introduces some significant new concepts. The existence and rapid expansion of interface protocols makes implementation of an actual demonstration of the megaprogramming concept feasible. Once a language-driven style of interoperation is established, some of the practical barriers which disable automation of large-scale computations will

have disappeared. CHAIMS can then also become a vehicle for new experiments in optimization and automatic programming.

9 Acknowledgments

The CHAIMS project is supported by DARPA order D884 under the ISO EDCS program, with the Rome Laboratories managing agent and by Siemens Corporate Research, Princeton, NJ. We thank Mehul Bastawala, Joshua Hui, Wayne Lim, Pankaj Jain, Mikel Pöss, and Catherine Tornabene for their contributions to the CHAIMS project.

10 References

1. Zelkovitz, M.V., *et al.*, *Software Engineering Practices in the US and Japan.* IEEE Computer, 1984. **17**(6): p. 57-66.
2. Libicki, M.C., *Information Technology Standards*. 1995. Digital Press.
3. Garlan, D., *et al.*, *Research Directions in Software Engineering.* ACM Comp.Surveys, 1995. **27**(2): p. 256-276.
4. Frakes, W. and C. Terry, *Software Reuse: Metrics and Models.* ACM Computing Surveys, 1996. **28**(2): p. 415-435.
5. OMG, *The Common Object Request Broker: Architecture and Specification*. 1991. Object Management Group (OMG) and X/Open. <http://www.omg.org/>.
6. Swartz, J., *OpenDoc, OLE squaring off*, in *MacWeek*. 1994. p. 2.
7. Chappell, D., *Understanding ActiveX (TM) and OLE*. 1996. 352 p. <http://www.microsoft.com/mspress/text/books/abs/1-216-5b.htm>.
8. JavaSoft, *Java(tm) Beans: A Component Architecture for Java*, . 1996. Sun Microsystems. <http://splash.javasoft.com/beans/WhitePaper.html>.
9. Rosenberry, Kenney, and Fisher, *Understanding DCE*. 1994. OReilly.
10. Miller, D.C. and J.A. Thorpe, *SIMNET: The Advent of Computer Networking.* Proceedings of the IEEE, 1995. **83**(8): p. 1116-1123.
11. Agarwal, S., *et al. Flexible Relation: An Approach for Integrating Data from Multiple, Possibly Inconsistent Databases*. in *IEEE Data Engineering Conference*. 1995. Taipei, Taiwan.
12. Courtney, A., *et al.*, *Inter-Language Unification, rel.1.5*, . 1994. Xerox PARC. <http://www.omg.org/>.
13. Wiederhold, G., P. Wegner, and S. Ceri, *Towards Megaprogramming.* Comm. ACM, 1992(11): p. 89-99. <http://www-db.stanford.edu/CHAIMS/>.
14. Wasserman, A. and S. Gutz, *The Future of Programming.* Communications of the ACM, 1982. **25**(3).
15. Booch, G., *Software Components with Ada*. Second ed. 1987. Benjamin Cummings.
16. Feldman, J.A., *et al. Recent Developments in SAIL.* in *Fall Joint Computer Conference (AFIPS)*. 1972.

17. Feldman, J.A., *High Level Programming for Distributed Computing.* Communications of the ACM, 1979. **22**(6): p. 353-368.
18. Xerox, *Interlisp Reference Manual*, . 1983. Xerox Corporation.
19. Pyle, I.C., *The Ada Programming Language*. Second ed. 1985. Prentice-Hall.
20. Mandrioli, D., *et al.*, *Modeling the Ada task system by Petri Nets.* ACM Computing Reviews, 1985. **10**(1): p. 43--61.
21. Summet, J., *Programming Languages*. 1969. Prentice-Hall.
22. Lorie, R.A. and J.F. Nilsson, *An Access Pattern Specification Language for a relational database system.* IBM Journal Research and Development, 1979. **23**: p. 286-298.
23. Tomlinson, C., *et al.*, *The Extensible Services Switch in CARNOT.* IEEE Parallel & Distributed Technology, 1993. **1**(2): p. 16-20.
24. Udell, J., *Component Ware.* Byte, 1994. **5**: p. 46-56.
25. Delis, A. and N. Roussopoulos. *Performance and Scalability of Client-Server Database Architectures*. in *18th Int. Conf. on Very Large Data Bases*. 1992. Vancouver, British Columbia, Canada.
26. McIlroy, D., *Mass-produced Software Components*, in *Software Engineering Concepts and Techniques*, J. Buxton and et al., Editors. 1976. Van Norstrand Reinhold.
27. Stroustrup, B., *The C++ Programming Language*. 1986. Addison-Wesley.
28. Meyer, B., *Reusable Software: The Base Object-oriented Component Libraries*. 1994. Prentice-Hall.
29. IEEE, *IEEE Standard 1278: Protocols for Distributed Simulation Applications: Entity Information and Interaction*. 1993. IEEE Computer Society.
30. Stickel, M., *et al. Deductive Composition of Astronomical Software from Subroutine Libraries*. in *Twelfth International Conference on Automated Deduction*. 1994. Nancy, France.
31. Musen, M.A., *et al.*, *PROTEGE II: Computer Support for Development Of Intelligent Systems From Libraries Of Components*, . 1994. Knowledge Systems Laboratory, Medical Computer Science, Technical Report KSL-94-60.
32. Vestal, S., *A Cursory Overview and Comparison of Four Architecture Description languages*, . 1994. Honeywell Technology Center: Minneapolis, MN. <ftp://src.honeywell.com/pub/ARCHIVE/dssa/papers>.
33. Tracz, W., *Confessions of a Used Program Salesman*. 1995. Addison-Wesley.
34. Smith, D.R., *KIDS: A Semi-Automated Program Development System.* IEEE Trans. Special Issue on Formal Methods, 1990. **16**(9): p. 1024-1043.
35. Papakonstantinou, Y., *et al. A Query Translation Scheme for Rapid Implementation of Wrappers*. in *International Conference on Deductive and Object-Oriented Databases*. 1995. <http://www-db.stanford.edu/pub/papakonstantinou/1995/querytran.ps>.
36. Perrochon, L., *Gateways in globalen Informationssystemen*, in *Department of Computer Science*. 1996. Swiss Federal Institute of Technology (ETH Zürich), Switzerland: Zürich. p. 155. <http://www.perrochon.com/>.
37. Birrell, A.D. and B.J. Nelson, *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems, 1984. **2**(1): p. 39-59.
38. ISO, *ISO Remote Procedure Call Specification ISO/IEC CD 11578 N6561*. 1991. ISO/IEC.
39. Boehm, B. and W. Scherlis. *Megaprogramming (preliminary version)*. in *DARPA SW Conference*. 1992: Meridien.
40. Samadzadeh, M. and M. Zand, eds. *Proc. ACM SIGSoft Symposium on Software Reusability*. . 1995. ACM.
41. Tracz, W., *Cost of Documentation in DoD*, . 1996. Personal communication (e-mail).
42. Jones, C., *Assessment and Control of Software Risks*. 1994. Yourdon Press.
43. Bell, R. and D. Sharon, *Tools to Engineer New Technologies into Applications.* IEEE Software, 1994. **12**(2): p. 11-16.
44. Parnas, D.L., *Designing Software for Ease of Extension and Contraction.* IEEE Trans. SE, 1979. **5**(2): p. 128-138.
45. Bach, J., *Process Evolution in a Mad World*. 1994. Borland Int, Scotts Valley CA.
46. Grady, R. and D. Caswell, *Software Metrics: Establishing a Company-Wide Program*. 1987. Prentice-Hall.
47. Lientz, B.P. and E.B. Swanson, *Software Maintenance Management*. 1980. Reading, MA. Addison-Wesley.
48. Blum, B.I., *Software Engineering - A Holistic View*. The Johns Hopkins University - Applied Physics Laboratory Series in Science and Engineering, ed. J. Apel. 1992. New York, NY. Oxford University Press. 588 p.
49. Newport, J.P., *The Growing Gap in Software.* Fortune, 1993. **113**: p. 32-142.
50. Mills, E., *Users fear hassles in using component software.* InfoWorld, 1994. **16**(38): p. 19-20.
51. Lowry, M.R. and R.D. McCartney, *Automating Software Design*. 1991. AAAI press/ MIT Press.
52. Charette, R.H., *Software Engineering and Risk Analysis Management*. 1979. McGraw-Hill.
53. Jackson, M.A., *System Development*. 1982. Prentice Hall.
54. Parnas, D.L., *Fighting Complexity.* IEEE TC EECS Newsletter, 1995. **2**(2).
55. Schneidewind, N.F., *The State of Software Maintenance.* IEEE Trans.on Software Engineering, 1987. **13**: p. 303-310.
56. Dertouzos, M.L., *The Multiprocessor Revolution: Harnessing Computer Together.* Technology Review, 1986. **89**(2): p. 44-57.
57. Finin, T., *et al. KQML as an Agent Communication Language*. in *CIKM 3*. 1994: ACM.
58. Michaelson, G., *Elementary Standard ML*. 1995. UCL Press. <ftp://ftp.cee.hw.ac.uk/pub/funcprog/gjm.book95.ps.Z>.

CHAIMS home page:
`http://www-db.stanford.edu/CHAIMS/`

ERROR: undefined
OFFENDING COMMAND: er

STACK: