BAA 95-40 EDCS

Technical Topic Area: Support for Continuously Evolving Software:

Technology Investigation at Stanford University

followed by options for

Technology Investigation jointly by Stanford and Industry

Title: Compiling High-level Access Interfaces for Multi-site Software (CHAIMS)

Technical contact: Professor Gio Wiederhold
Computer Science Department
Stanford University
Stanford CA 94305
phone: 415 725-8363; fax: 415 725-2588
email: gio@cs.stanford.edu

Administrative contact: Marianne Siroker
Computer Science Department
Stanford University
Stanford CA 94305
phone: 415 723-0872; fax: 415 725-2588
email: siroker@cs.stanford.edu

Total Cost:	FY1996	FY1997	FY1998	Total requested
Base Total:	$ 392,187	$381,696	$393,452	$ **1,167,335**

Estimates of options for further work, to be negotiated later

	FY1996	FY1997	FY1998	FY1999	FY2000
Option 1:		$102,000	$204,000	$210,000	$110,000
Option 2:		$134,000			
Option 3:			$400,000	$400,000	$400,000

Type of Business: Other Educational: Non-Profit Private University

prepared 8 Nov. 1995, submitted Nov. 1995, minor edits 8 May 1996.

A: Innovative Claims.

The CHAIMS effort will develop a very-high level (*mega*-) programming language for software module composition. Its compiler will generate a variety of invocation sequences for current and developing standards for software interoperation, with a focus on multi-computer, distributed operation. The language will include the ability to set up module interfaces prior to executions, request performance estimates from modules prior to their invocation, schedule modules in parallel, monitor execution of invoked modules, interrupt inadequately performing modules, and provide data and meta-information to customer interface modules [WiederWC:92].

CHAIMS supports a paradigm shift which is already occurring: a move from coding as 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. There are hence few tools and inadequate education 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. CHAIMS intends to overcome the gap in composition tools.

Many programming languages have had some composition facilities added to their basic capabilities, making them unwieldy. The CHAIMS mega-programming language will only serve module composition and scheduling and remain simple, although it has to introduce some significant concepts. A focus on the issues critical to system integration will allow a high capability to be attained without superseding important facilities provided by legacy and evolving languages. The size of the modules we envisage 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 a dynamic, multi-site setting. Such interfaces are specified by interoperation standards as CORBA, DCE, and KQML. These protocols provide, in effect, our machine languages and make the CHAIMS concept implementable today, whereas it was not, say, three years ago.

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

1. Flexibility of interface standard choice, changeable late by only megaprogram recompilation.
2. Composition clarity by making instances of domain-model architectures obvious.
3. Testing of new and updated software without committing to specific configurations.
4. High system performance since asynchronous and parallel operation is the default.
5. Life-time maintenance as a seamless continuation of system development.
6. Adaptability of applications to evolving interface and product standards.
7. Optimization and adaptation to change of scale, modules, and network configuration.

Each of these claims is elaborated below in Section D. However, 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 mega-program, individual module performance can be improved by allowing a choice of platforms and module programming languages.

B: Deliverables.

This proposal is innovative in the sense demanded by the EDCS BAA 95-40. To track progress it will be executed in a sequence of prototype development cycles; there are deliverables with each cycle. Beyond the base cycles are some options. The central deliverables, a programming language specification and an open, prototypical compiler for the language provide a tool that enables the composition and lifetime goals central to large-scale, usable and evolvable software systems [AdlerW:92].

We do not expect to invoke proprietary claims on the language or its prototype compiler at Stanford. We will publish the results of each cycle, including metrics obtained, rapidly on the worldwide-web, and subsequently in formal form. The most effective form of dissemination we can envisage are our graduates moving to companies that are not risk-adverse. Commercialization in subsequent cycles by industrial partners may be proprietary. We defer negotiations on industrial protection of intellectual property to such time that Option 3 for joint Technology Investigation are exercised.

Specific Deliverables, based on an initiation of funding 1 July 1996:

Cycle 1: 31 Dec.1996: Specification of the CHAIMS language.

Cycle 2: 30 June 1997: CHAIMS assessment, refinement and definition of metrics. Specifications of a KQML-module interface,

Cycle 3: 31 Dec.1997: Initial CHAIMS implementation and initial prototype with KQML connectivity. Assessment of functionality and qualitative metrics.

Cycle 4: 30 June 1998: Demonstration of CHAIMS capabilities as multi-site operation and scheduling. Operation with either a CORBA, DCE, or OLE interface. Implementation of a KQML-module specification repository. Performance parameters.

Cycle 5: 31 Dec.1998: CHAIMS multi-site optimization performance assessment and quantitative metrics. A CHAIMS/KQML API

Cycle 6: 30 June 1999: CHAIMS architecture guideline. Updated software to comply with the most recent versions of interoperation interface standards. Evaluation of CHAIMS/KQML interface in a distributed setting. A predictive performance model for repository access and operation. Comprehensive CHAIMS final prototype documentation.

Option 1: Stanford Logic Group (Based on a start date of 31 Dec.1996):

Cycle 1.1: 31 Dec. 1997: Implementation of on-line glue code generation for KQML modules in support of CHAIMS

Cycle 1.2: 31 Dec. 1998: Implementation of off-line glue code generation for KQML modules (matchmaking).

Cycle 1.3: 31 Dec. 1999: Multi-site optimization of matchmaking services for KQML modules in support of the CHAIMS compiler.

Option 2: Stanford Database Group (Based on a start date of 30 June1997):

31 Dec.1997: Documented framework and component modules for composition metrics.

Option 3: Industrial partnership (Based on a start date of 30 June1997):

31 Dec.1997: Industrial transition of CHAIMS prototype. Platform selection.

30 June 1998: Verification of re-development plans. Resource commitment.

31 Dec.1998: Alpha-version delivery to a potential user site.

30 June 1999: Beta-version delivery. CHAIMS product documentation.

31 Dec.1999: Establishment of formal end-user support with a help-desk.

30 June 2000: Hardened CHAIMS Compiler, documentation, end-user support.

C: Statement of Work.

We will develop, assess, and demonstrate a mega-programming language, CHAIMS, whose function is to support integration of existing modules. The modules themselves may use a variety of interface standards as promulgated today, as CORBA, OLE, KQML, DOE, DCE, PCTE, etc. The binding of standards selection of interfaces can be delayed arbitrarily, and changed by recompilation of the CHAIMS program. The proposed language will allow composition of such modules operating on a variety of platforms in a distributed network of computing resources. Again, recompilation of a CHAIMS program will allow network configurations to be changed. We will not investigate means of network independence of generated mega-codes [MerksDRDC:92], since recompilation should be easy, and may require rescheduling of operations for global efficiency, a task which is beyond the capability of any of the adaptive schemes that we have seen [FriederS:91].

The components of CHAIMS are symbolically depicted in Fig.1. The topmost component is the (mega-) programmer, who selects the modules to be invoked.
The mega-program, written by the mega-programmer, only manages the invocation of the modules, according to precedence constraints that typically represent dataflow dependencies. Any needed conversions to be performed in the dataflow are invoked externally to the mega-program, assuring that the mega-program is a clean representation of the architectural intent for this application. The mega-program is compiled by the CHAIMS compiler, the resulting executable program consists mainly of module invocations, expressed using the interface standard protocols that are appropriate for this module. Many of these invocation 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 [Tracz:95].

- Mega-programmer: defines architecture
- Mega-program: schedules the invocation of modules
- Standard client-server interfaces
- Modules: preexisting and written to order

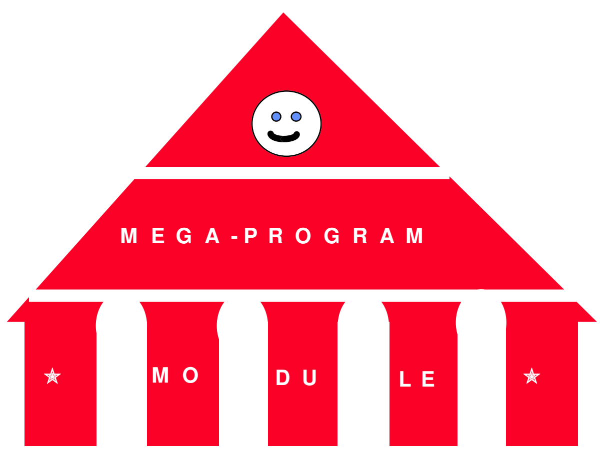

Figure 1: Components of the CHAIMS mega-programming paradigm.

The process of the CHAIMS mega-program generation and use is illustrated in Fig.2. Note that this proposal does not envisage automatic programming or knowledge-based techniques [Smith:90]. It simply provides a new high level tool for module composition. Neither does it provide novel interfaces. The result of a mega-program execution are simply shipped to an output module, written again in any language that is compatible with the end-user or customer platform. However, once a language-driven style of interoperation is established, some of the practical barriers which disable automation of large-scale computations will have disappeared.

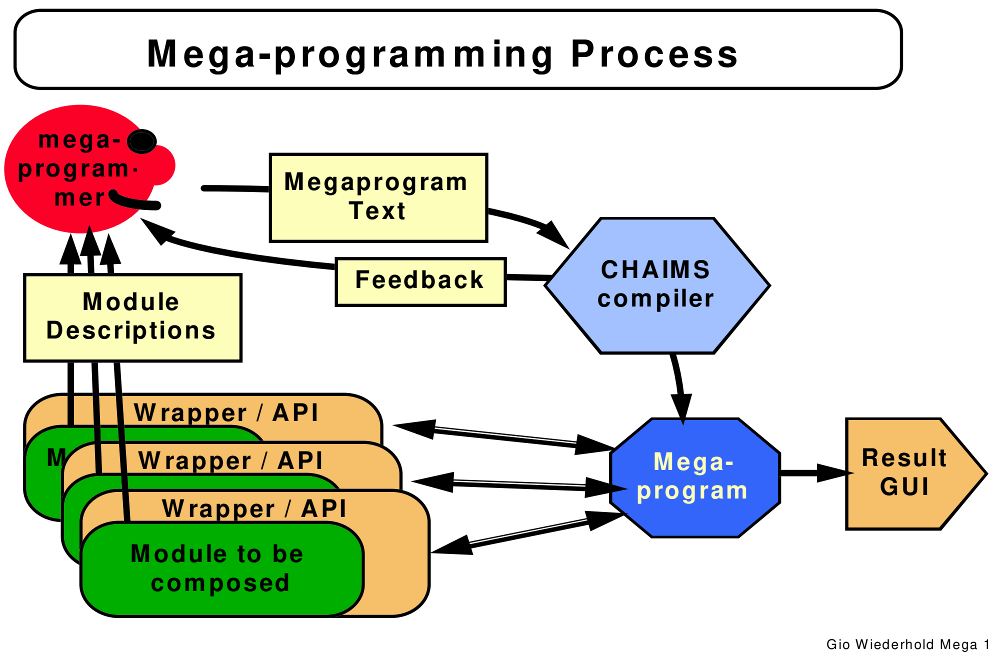

Figure 2: The CHAIMS Mega-programming Process.

The modules to be composed by CHAIMS can be written in any language that is able to create the required interfaces, or that generates code that can be suitably wrapped. We certainly will demonstrate modules in C and C++. The FORTRAN and Ada languages are high on our list as well. We intend to profit here from the many efforts in supporting interoperation interfaces currently in process, and likely to be accelerated by other proposals in the EDCS suite. Where required, glue-code to resolve semantic differences will be supplied by modules written in any suitable programming language that can generate a compatible interface. The CHAIMS compiler itself will not be burdened with having to generate such glue code, since here conventional languages perform already well.

The CHAIMS language will be designed to solely serve system integrators, an increasingly large fraction of the software community. CHAIMS will not obsolete specific languages to write modules, although its use will enable new module efficiencies due to greater flexibility in making platform assignments and specialization. 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.

KQML as a Model Interoperation Language. One of the languages we plan to investigate in CHAIMS is the Knowledge Query and Manipulation Language (KQML).
KQML has features not yet found in many earlier languages for interoperation and will allow stress-testing of the CHAIMS approach. A version of KQML is being proposed to OMG as a successor to CORBA 2. KQML is a part of the ARPA Knowledge Sharing Initiative, and sets a framework for communicating programs as well as data between components. A KQML expression involves a sender and receiver, and it makes explicit its content (e.g., a program fragment) and the nature of the communication (execute this program, save this program, etc.). An important feature is the ability for components to provide specifications of their capabilities. This will provide a useful service in support of the CHAIMS compiler and mega-programmer. A prototype interpreter for KQML has been developed at Stanford in the ABSI project [GenesereththSS:94] and forms the basis for supporting the CHAIMS compiler for KQML software modules.

Testing and Life-cycle support. A significant advantage of a language-based approach to software composition is the capability to provide feedback to the mega-programmer. The interface protocols that are currently available to build client connections to servers are often awkward, and writing them correctly is hard. Without a language that understands these protocols any errors, outside of coding syntax errors, cannot be reported until execution time. Testing of integrated systems is especially costly for distributed computations. Because setting up realistic multi-site execution scaffolding environments for testing is expensive it is often omitted. As a consequence no timely feedback is obtained prior to a system integration phase, which, in turn, leads to very high final test and debug costs at integration time [Rettig:91].

Testing of fully deployed systems also has extremely high risks. Since the mega-program remains the active essence of the client application, rather than a design artifact, it also provides the basis for life-time maintenance. Module changes can occur continuously, but any interactions with other modules, any architectural adaptations, and any reassignment of functions require mega-program maintenance. That means that true isolation and module encapsulation is achieved. In deployed systems modules can be isolated for maintenance, and carefully verified in isolation. A capability to update modules incrementally reduces pressures to aggregate many module updates into one large and risky system update. Recall that the objective of CHAIMS is not to do less maintenance, but to do it better, with fewer errors and more rapid responses to customer needs.

D: Description of Expected Results .

We expect to demonstrate and analyze the benefit of using a mega-programming language for the development and ongoing management of multi-site software systems. We believe we can succeed now because we can build on the efforts that have been expended in the development on interfaces for interoperation within client-server models [DelisR:92]. We do not expect to develop new interfacing standards, but instead demonstrate their utility (and perhaps their weaknesses) as primitives for a mega-programming language, moving to a new programming paradigm [Kapoor:91]. We will show new ways to utilize these interfaces by developing a partitioning of the module invocation, breaking up the function of the traditional `CALL' statement, specifically in the context of Remote Procedure Calls (RPCs). Technical specifics are provided in Section F.7.

To encourage technology transition we will work with small software companies as transition agents. We believe that such companies are much more effective than universities can ever be in technology transition, since they can provide the proper long-term career path for software professionals, including our university graduates [WiederholdY:95]. Option 3 focuses on the technology transition aspect of our proposal.

Specifically, we will focus on the advantages of the approach proposed for CHAIMS as listed here:

1. Flexibility prior to actual deployment. The delayed binding provided by CHAIMS relaxes commitments to specific interface standards. Choices among standard interfaces as CORBA, OLE, DCE can be deferred until the scale of the problem being addressed is known and the platform capabilities can be assessed. This flexibility goes beyond the facilities that programming languages can provide in terms of platform allocation [Newport:93].

2. Composition clarity. The structure of a mega-program 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 [Mills:94]. As such, it supports the concept of domain-specific software architectures (DSSA). Clarity of architecture, and autonomy of module development attacks one of the major problems leading to system failures [Charette:79].

3. Testing. Since the logic of a mega-program can be tested in other, and simpler environments than the eventual implementation it should be possible to commence testing early in the development cycle. Rapid prototyping is served. CHAIMS also permits to show to potential customers realistic prototype operations covering a valid architecture, using simple modules and economical platforms, avoiding costly waste due to implementation of unsuitable design choices [Boehm:84]. The capability of CHAIMS to deal with modules that export performance estimates enables prototypes to provide system performance estimates before systems are fully populated.

4. Asynchrony. Parallel operation of the mega-modules is the default in CHAIMS. This vision means that the increasing availability of distributed computing can be exploited without resorting to parallel computing constructs. By considering any sequential dependency as an exceptional constraint the mega-programmer 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. It is actually hard to visualize how crucial DoD tasks, as battlefield simulations, can be effectively described in a sequential program.

5. Maintenance. The major economic benefit of tools to manage the mega-process is in support of system maintenance. In today's practice, even where design and implementation of software are done to high professional standards, the tools and documents used in that process are not available in an effective, automated manner to the maintainers after software installation [Schneidewind:87]. Since the cost of system maintenance today is about 60 to 80%, and this cost increases disproportional with size of the systems, it is in maintenance where the leverage of mega-programming is greatest [Basili:90]. Since the mega-program represents the executable specification of the architecture, there is no gap between the instance of the application architecture and the execution. The same objective holds for *Executable Specification Languages* (ESL), but these focus on successive refinement of specifications into a program. Once the program has been developed all the way from the ESL the inverse path may be hard to retrace. Note that module changes will affect the behavior of a composed system. If the changes are such that the exposed interface, as determined by the parameters used to SET-UP input and EXTRACT results from a module, are not affected, then we have reason to assume that the mega-programmer need not be concerned with the change, since the mega-program is at a higher level of abstraction. On the other hand, if module changes are exposed through these parameters, then the mega-programmer does get involved and can decide to keep using an older module or adapt to the new version of the module. However, autonomy is achieved among modules, greatly simplifying maintenance.

6. Adaptability to changing standards. A unique aspect of distributed programming is that it depends on interface standards that themselves are undergoing rapid evolution [Libicki:95]. The instruction codes that defined a family of computers remain fairly constant over its lifetime, so that the code generation portion of code compilers needs little maintenance, once it was comprehensive and fault-free. The interfaces for distributed computing are likely to stay fluid. For instance, SQL is undergoing major 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 1. The protocol for the web, HTML is growing in several directions simultaneously, and KQML has *classic* and recent versions co-existing. Isolation of the mega-program designer from these changes, and allowing recompilation of CHAIMS programs once new interface-code generators have been validated, is a major benefit of the mega approach presented here.

Recognizing changes that affect interaction semantics remains a problem. The first option proposed addresses the problem of semantic mismatch between modules. Our effort will first focus on KQML modules [FininWa:94] and their integration with the CHAIMS compiler. The semantic mismatch spans the spectrum from a simple mismatch in vocabulary between a calling module and a called module (e.g., simple type conversions for arguments) to more extensive translations to transform a call into a collection of simpler calls, which can be handled directly by existing modules. The semantic mismatches found are resolved by generating glue code for modules that are to be connected. This glue code is generated off-line based on the specifications of the modules. Additional information is provided in Section F.10.

7. Optimization. There are two aspects to distributed optimization, and CHAIMS will provide a basis for innovative development of both of them: (a) optimal ordering of module invocation, i.e., the control flow, and (b) optimal transfer of data among modules, i.e., the dataflow.

7a. Optimal module invocation and adaptation to change of scale.. Use of predefined objects and modules allows rapid composition, but criteria for performance are effectively encapsulated within those objects and modules. Any scheduling to reach an acceptable level of performance depends on the programmer's understanding of the hidden structure of these objects and modules, a very undesirable state of affairs. Even if programmers guess right, as is likely in in initial design, they will be wrong after a few years when modules and configurations have undergone maintenance and scale changes. Automatic optimization is similarly disabled if schedulers cannot model the performance of system components. In the CHAIMS approach the *modules themselves provide their performance estimates* via distinct entry points, enabling the CHAIMS scheduler to re-optimize systems as the configuration and usage parameters change [Wiederhold:92o]. Parameters to be supplied by the modules are principally execution cost, time, and result volume, with uncertainties for each [WhangKW:94]. The splitting of optimization into a CHAIMS scheduler and module provided parameter generation addresses a major scalability problem, which today is commonly ignored or delegated to post-delivery tuning.

7b. Optimal data transfer over networks. Since CHAIMS execution is expected to take place within a network configuration, the scheduler also has to take into account parameters that are system dependent, namely bandwidth and latency of connections. Optimal module distribution in network configurations has been described, but as far as we know there never has been a tool in which such optimization can be performed to the degree that executable code is generated. Within this proposal we only expect to explore existing optimization algorithms for networks, some of them based on fluid-flows. The framework provided in CHAIMS should enable others to test the effectiveness of their ideas as well.

After compilation, the module controlling the mega-program and its communication links will be relieved of much interaction since high-bandwidth data flows can now move directly between modules. The result of such a compilation, as illustrated in Fig.3, has a close relationship to typical inputs for a dataflow computer, and algorithms devised to optimize dataflow may well be applicable for further refinement of such a distributed computation. It is clear that direct communication reduces total bandwidth demands, and also reduces total execution times due to latency reduction gained by bypassing the mega-program for data transfer.

The formal separation of the mega-program and its modules, supported by the distinct function of the CHAIMS partitioning of CALL statement functions, provides a natural firewall. For instance, the ESTIMATE sub-statement detailed in Section F.7 removes the programmer's need to break module encapsulation in order to get the information needed for adequate performance.

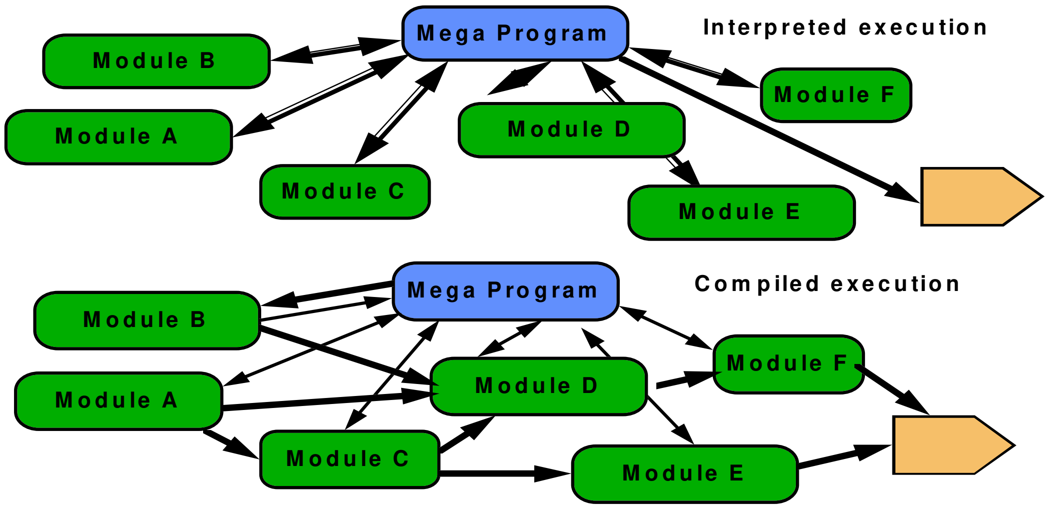

Figure 3:
Dataflow of a Distributed Computation after Optimizing a CHAIMS Mega-program

Examples of DoD software systems where optimization of dataflow is crucial include distributed simulation, battlefield data distribution, and concurrent manufacturing. In all those instances optimization is performed today at the initial design phase, and hard to change subsequently.

Expected Results for Options:
We include three options for extending this proposal beyond the base.

Option 1: 1 Jan. 1996 -- 31 Dec.1999.
Off-line generation of glue code based on module specifications.
Individual programmers using CORBA and similar interface languages create modules without knowledge of the data structures and algorithms of other modules, and without knowledge of the hardware configuration in which the module will run. However, in these systems, the module interface descriptions are in an Interface Description Language (IDL), and the meaning of the IDL interface cannot be defined in IDL (e.g., the fact that a given interface sorts a list). The meaning of the service provided by a module must be known by external means, and there is no support for standardizing or defining the semantics of the interfaces. This requires manual coordination between programmers in a standard CORBA-like environment to agree on the meaning of interfaces. In a large setting, no human can have such global knowledge, and the traditional coordination approaches break down. A declarative interface addresses these issues directly by providing a fixed semantics and by providing a mechanism for examining and defining the vocabulary used.

In order to link modules glue code will be needed. The glue code will be responsible for resolving semantic mismatches between different KQML modules that are integrated in a mega-program (see section F.10). We expect that this option will incrementally develop a toolbox for semantic matchmaking, which can be invoked manually or automatically by the CHAIMS compiler. The CHAIMS compiler will be enhanced to invoke glue code to resolve semantic mismatch of modules. This option will bring CHAIMS closer to the ideal of hyper-programming.

Effort: 50% Research Associate, 50% programmer, 1 student (50% academic year, 100% summer).

Option 2: 1 July 1997 -- 31 Dec.1997:
Metrics Evaluation Base Package:

As discussed in Section F.9 below, we prefer testing our approach with shared industry metrics. Since they do not exist now, we propose a small options to establish and document modules suitable for measuring progress in composition research by us and by others. This option can start after the initial research has provided an adequate understanding of potential metrics. The result will be a framework and component modules for composition metrics.

Effort: 10% PI, 40% postdoc, two students (100/50% each)

Option 3: 1 July 1997 -- 31 June 2000.
Technology Investigation with Industry:

To achieve effective mega-programming tool and technology we will contract with a small software company to take over advanced development, system integration, introduction, marketing, sales and long-term service tasks. We believe that such companies are the best means for technology transition to industry. They, in turn, will interact with end-users. A university is not able to serve end-users, since student and faculty priorities are different. Adding long-term staff for such tasks to university projects is unwise, since research projects are by their very nature cyclical [WiederholdY:94].

The option can also start one year after CHAIMS research is initiated, and should provide benefits both to the industrial partner and to Stanford researchers. Funding is needed to achieve expedited transition, since this direction is too novel to satisfy venture investors or research departments of large companies. If this option is funded, we will document and announce our technology widely to potential partners, as listed in Section F.11, and select one or more. Input from ARPA will be helpful in those negotiation, to assure relevance to DoD. We are, of course, also willing to work with companies that are funded independent of this contract. (see Section F.11). The expected results will include a hardened CHAIMS mega-program compiler, documentation, and end-user support

Effort: 20% PI, 40% postdoc at Stanford, $300K/year subcontract to industrial partner.

E: Cost, Schedule, and Milestones of the Base CHAIMS Proposal.

Cycle 1: 1 July 1996 -- 31 Dec.1996: Specification of the CHAIMS language, selection of a sample application and a process scenario.
Logic Group: Specification of CHAIMS/KQML-module interface.
Result: CHAIMS and KQML-API specification.

Cycle 2: 1 Jan. 1997 -- 30 June 1997: CHAIMS assessment through expert interaction and early implementation. Definition and development of some relevant modules.
Logic Group: Implementation of an initial prototype API using KQML.
Result: CHAIMS refinement, experimental base, and definition of metrics.

Cost for the period from 1 July 1996 to 30 June 1997: $392,187.-

Cycle 3: 1 July 1997 -- 31 Dec.1997: Experimentation with the initial CHAIMS implementation. Assessment of functionality. Development and demonstration of a second application scenario.:
Logic Group: Implementation of the CHAIMS/KQML API..
Result: CHAIMS functional assessment and qualitative metrics.

Cycle 4: 1 Jan. 1998 -- 30 June 1998: CHAIMS architecture development, including multi-site operation and scheduling principles. Module interface augmentation to provide performance parameters. .
Logic Group: Implementation of KQML-module specification repository.
Result: CHAIMS refinement, repository access, and definition of metrics.

Cost for the period from 1 July 1997 to 30 June 1998: $381,696.-

Cycle 5: 1 July 1998 -- 31 Dec.1998: Implementation and assessment of multi-site optimization. Experimentation with various configurations. Assess performance.
Logic Group: Evaluation of CHAIMS/KQML interface in a distributed setting.
Result: CHAIMS performance assessment and quantitative metrics

Cycle 6: 1 Jan. 1999 -- 30 June 1999: CHAIMS architecture guideline development. Interaction with industry. Updating to most recent versions of interoperation interface standards. Development of a predictive performance model.
Logic Group: Optimization of KQML repository in a distributed setting
Result: Comprehensive CHAIMS final prototype documentation. Guidelines for system optimization.

Cost for the period from 1 July 1998 to 30 June 1999: $393,452.-

Total cost for the period from 1 July 1996 to 30 June 1999: $1,167,335.-

F: Technical Rationale and Technical Approach.

We summarize now the reasoning behind the CHAIMS approach; these points will be elaborated later in this section:

1. There is a shift in the software generation paradigm from an emphasis on programming as code generation to an emphasis on composition and integration, typically distributed over many nodes on high- or mixed-performance communication networks [ZelkovitzA:84].

2. There are now 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 [Libicki:95].

3. Rather than waiting for the final comprehensive interoperation standard (as if it were Godot), we should be independent of any specific standard. We need now a high-level language to overcome interface and network architecture differences and gain independence for large-scale applications. Independence was achieved for hardware first by programming languages as Fortran IV, Lisp 1.5, Cobol, and Ada. Integration today is still platform dependent, even when languages used to program the software are not [GarlanA:95].

4. The focus of the proposed mega-programming 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 [Wiederhold:95]. Low cost incremental maintenance will also increase the responsiveness to change requests.

This proposal does not yet reach the vision of hyper-programming presented in the EDCS BAA, where formal program understanding, even understanding derived from legacy programs, drive the maintenance and re-engineering processes. Understanding in our model is still focused on the mega-programmer. Modules have to be adequately documented and perform predictably in their contexts. The strict encapsulation planned for CHAIMS modules, and the ability to gain information from them without breaking the encapsulation (see Section F.7), provide a significant stepping stone on the path to hyper-programming.

F.1 Motivation for seeking government and DoD funding.

Since the broad concept of a protocol-independent mega-programming language does not favor any specific company or consortium, it is unlikely that any existing enterprises can justify the research and development investment proposed here. Large distributed and adaptable software is crucial to DoD. Current software delivered to DoD is difficult to maintain, and contractors have few incentives to deliver better software

[Karr:93]. The requirements in DoD that make Ada a primary choice have to be balanced with commercial resources that are written in other languages. CHAIMS provides graceful interoperation of mixed module configurations.

The CHAIMS approach focuses on the task of system integration. Ten years ago, integration of large-scale software was performed mainly by experienced groups in large companies, as in IBM, UniSys, Fujitsu, Arthur Andersen and the other big five financial system houses, and by system contractors as SAIC, Lockheed, MITRE, Lincoln Labs, etc. Commercial companies, competing on a price basis, had strong motivation for reuse, but the government and DoD contracts were often based on cost, reducing incentives to invest in reusable modules [Kruger:92]. Today, software composition has moved to UNIX and PC platforms, and an increasing fraction of the software workforce is engaged in composition, as sketched in Fig. 4 [BarrT:1995]. At the same time, DoD wants to move towards acquisition of Commercial-off-the-shelf (COTS) products, but two factors hinder progress: (1) lack of tools and (2) the need to interoperate with specific defense-specific modules.

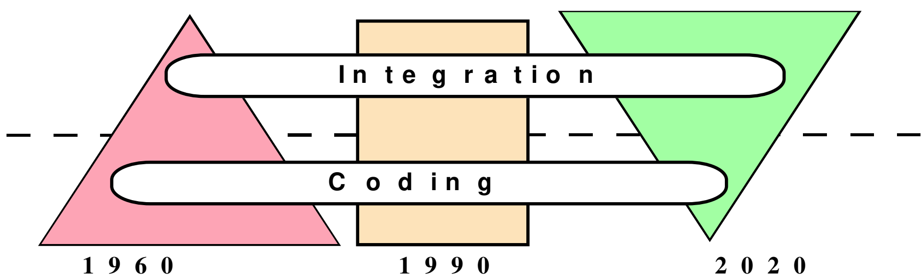

Figure 4. Re-allocation of the Workforce in Computer Software Construction

We cannot expect that CHAIMS will provide the only solution for large-scale software interoperation, but a demonstration of advanced concepts is needed now. The strong trend in the direction of composition will provide a broad industrial audience for the proposed research [GarlanA:95]. Having an example of operational mega-programming will bring the discussion of the facilities, technologies, and standards to a high level of abstraction, while maintaining a solid footing in terms of implementation and utility [Wiederhold:93].

F.2 Tools to advance the development and evolution of large-scale software

There are two aspects of mega-programming, *process* and *tools*:

1. 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 [BoehmS:92]. Reusing `second-hand' modules takes advantage of the validation conferred by practice [SamadzadehM:95]. 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 60% of SW generation costs.

2. Tools to achieve rapid composition is a largely ignored area, except for meta tools. Meta tools 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 glue which manages large-scale, distributed software as it must evolve over time [Parnas:79].

The CHAIMS proposal here focuses on the second aspect: **tools**. Having general, productivity-enhancing tools simplifies adherence to the mega-programming process. Tools also provide an opportunity to measure adherence directly, whereas today adherence to the process is measured by collecting documents and signoffs from the programmers and their managers [Bach:94]. Furthermore, and economically more crucial, a mega-programming language as CHAIMS, and hence any assessment tools bound to it, will stay with the system beyond delivery, throughout the maintenance phase, where 60 to 80% of system costs are incurred and where also the majority of errors are introduced. This aspect is further discussed below in Section F.4: Lifetime of software systems.

The mega-program 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 Domain-Specific-Software-Architecture (DSSA) [LowryM:91]. The architecture instance defined in a mega-program is reusable, not only 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 mega-program will be limited by the capabilities of current interface languages, but these are improving rapidly.

In the scope of this proposal we do not intend to provide a graphic editor to manage a descriptive boxology of the mega-program's architecture. Many such tools are available, e.g., [Jackson:82]. If we reach a state that the mega-program becomes so large that it is hard to understand, such tools can be adapted. We hope that the mega-programming approach advocated limits complexity so that such tools, are not needed [Parnas:95]. Validation of a CHAIMS architecture instance is through its execution, including the execution of the mega-program in smaller environments.

F.3 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 [Tracz:95]. When the focus is on development, the case for investment in reusable modules is indeed hard.

The CHAIMS proposal 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 [Schneidewind:87]. Development tools can reduce initial costs and improve products, but are rarely designed to carry through into the operational and maintenance phase [Luqi:90]. It is in long-term maintenance support that we can make the greatest contribution and have most leverage, and this is a natural outcome of using a language that generates executable code. If maintenance cost (not the amount of maintenance) can be reduced by 25%, we can double investment in true innovation. If on the other hand, the increase of scale leads to larger relative maintenance cost, we can lose all ability to innovate. Many of us are familiar with enterprises where innovation has been disabled by the rate of unscheduled software efforts.

As described in Section D, maintenance consists of bug-fixing, adapting to changing environments, and perfecting. Our hypothesis is that, once the basic architecture of a system is established, all three maintenance categories generally require only local, that is module-specific changes. The only communication among modules is via the mega-program, and the mega-program communicates variables only via SET-UP and EXTRACT parameters (see Section F.7). Those parameters are accessed and described via associated meta-statements [WiederholdWC:92], so that the exposure is rigorously controlled. This limited exposure greatly reduces the semantic risks. We know of no way to formally manage semantics of large-scale programs, but reducing their exposure so that their effect is reduced will have obvious benefit in maintainability. Both the mega-programmer and the module provider lose some control, but gain autonomy in the process. If our hypothesis holds, then this gain in autonomy will far outweigh the losses in efficiency obtainable through close-knit integration of modules within a system.

We are concerned with system-level optimization, and hence provide the ESTIMATE and INSPECT substatements, as detailed below in Sections F.7 and F.8. Global optimization and interaction is now no longer a valid reason to break the encapsulation of the module-sized objects and create maintenance nightmares [Wegner:95].

F.4 Lifetime of software systems.

The high cost of software maintenance is largely due to its long life time [Wiederhold:95]. Abstract process models become irrelevant to operational needs [MarcoB:87]. Quantitatively, maintenance consists of bug fixing (20%), adaptation to changing environments (laws, rules, policies, enterprise reorganization: 25%) and perfection, namely adding minor functionalities in order to keep customers (55%) [Lientz:80]. The
need and cost to adapt is probably greatest in large software systems, as seen in DoD, since the number of external interfaces increases. The low fraction of bug-fixing is due to the long lifetime of software, ranging widely, but on the average about 15 years. Bugs are an issue early in the lifetime of software, but in software that survives, any bugs that remain after one year are ignored and bypassed [BarrT:95]. Computer hardware is often replaced after three years, leading to high depreciation and low maintenance. When

hardware is replaced, being able to use or adapt existing software is the major concern. Software, in contrast, is characterized by low depreciation and high maintenance costs.

Since the mega-program is an actual part of an application, it can 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. We can only expect that overall interface protocols will become more powerful, so that limitations we will experience initially when using DCE, CORBA, or OLE will be mitigated over time.

F.5 Standards for interoperation.

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. 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 [Dertouzos:86].

Interoperation standards provide the infrastructure for the mega-programming 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 the m in a correct and efficient manner. Execution of multiple modules and the mega-program itself occurs in parallel, unless prohibited by explicit constraints. The modules themselves may use a variety of interface standards as promulgated today, as CORBA [OMG:91], DCE [Lockhart:95, OSF:91], DIS [IEEE1278:93], DOE [AgarwalKSW:95], ILU [CourtneyJB:93], KQML [NechesFFGPSS:93], OLE [Barnhart:94, Brockschmidt:94], OPENDOC [Swartz:94], PCTE [ECMA:92], etc. The binding of standards selection of interfaces can be delayed up to execution time, although compiled CHAIMS mega-programs may have to be recompiled to adapt to new interface standards.

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 software in large systems, which may involve multiple computers of differing classes. The interoperation standards are evolving rapidly; some are bound to grow and others will diminish in importance. Just as traditional programming languages provide an insulation from platform differences, the essence the CHAIMS proposal is to provided insulation 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.

F.6 A language solely for composition and interoperation.

Many programming languages have had some composition facilities added to their basic capabilities, making them unwieldy. Some information systems have had dedicated languages, as databases (SQL), report generators, and distributed information systems [TomlinsonCMW:93]. The Microsoft environment has had success with Visual Basic [Udell:94].

The CHAIMS mega-programming language will **only** serve module composition and scheduling and remain simple, although it has to introduce some significant concepts. Focusing on the issues critical to system integration will allow a high level of capabilities to be attained without conflicting with important facilities provided by legacy and evolving programming languages [Arnold:93]. The size of module we envisage typically justifies a dedicated processor, although modules can share a single processor when performance demands permit. We expect to support modules written in C, C++, Ada, Fortran, although they typically will need interfaces (similar to APIs) to allow interoperation in a dynamic, multi-site setting [Andrews:93]. We expect to support interoperation standards as CORBA, DCE, and KQML, and remain open to others. The existence and rapid expansion of these interface protocols makes the mega-programming concept feasible today, whereas it could not have been feasible, say five years ago, when the conceptual paper for our mega-programming language was written [WiederholdWC:92].

The interoperation protocols, developed as part of the thrust to move to client-server applications (see Section F.5 above), provide in effect the machine language for CHAIMS compilation.

F.7 Unloading the <u>CALL</u> statement.

The subroutine CALL statement, established in 1956 as a means to invoke code fragments with parameters (prior to that invention parameters and return addresses needed to be left in pre-designated memory locations) has seen a growth that closely mirrors the growth of the scale of computing overall. Today generic CALL statements invoke methods hidden in objects of various types, and Remote Procedure CALLs (RPC) transport data and results over networks. We see specifications for procedure calls with dozens or more parameters, of mixed types. Some languages specify distinctions of input, output, and in-out parameters. Others permit both positional and labeled parameters to reduce the confusion while permitting acceptable performance. Some system specifications require parameter blocks, lists of parameters transmitted as single units, that are then shared by caller and callee.

We are convinced that the CALL statement has become overloaded [WiederholdWC:92]. A prime contribution of CHAIMS to large-scale module composition is the splitting of the CALL statement into 5 substatements. Such a split would induce intolerable overhead into simple procedure invocation, but actually reduces invocation cost for major modules, because the 5 substatements can be independently executed and re-executed, reducing the overhead that comes with monolithic CALL invocations.   Fig. 5 illustrates the concept.

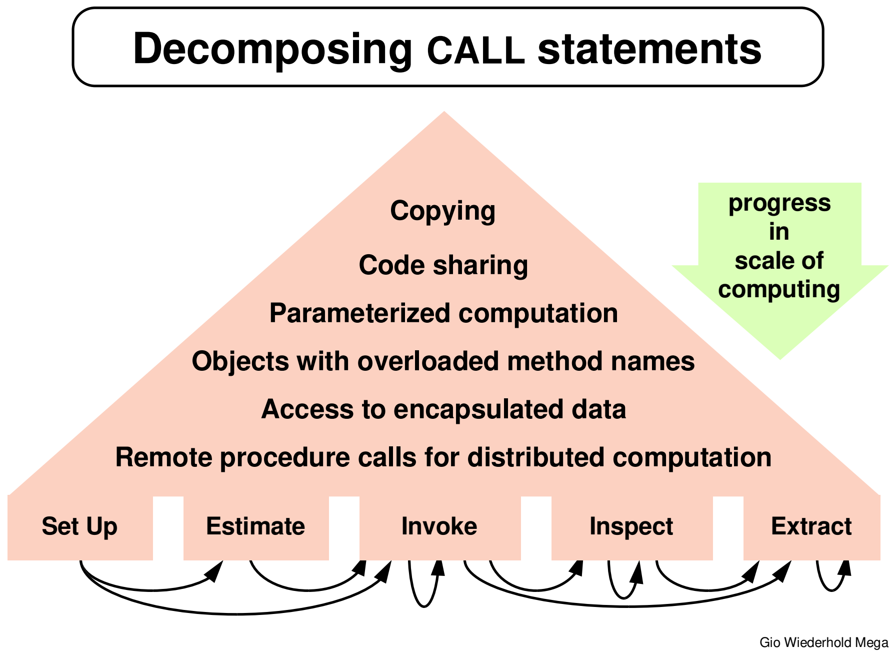

Figure 5: The Decomposition of the Procedure CALL statement in a Historical Context.

1. The **Set-Up** substatement supplies the bulk of the parameters, and allows the module to prepare itself for efficient execution For instance, a travel planning module can obtain from its databases data on the likely regions that will be involved in the planning activity. Optimization will attempt to move SET-UP statements as much as possible in advance of module invocation.

2. The **Estimate** substatement obtains meta-data from the modules permitting global mega-program optimization. Its role is elaborated in Section F.8.

3. The **Invoke** substatement initiates the actual module execution in parallel, and returns to the mega-program. It can be repeatedly executed if major set-up parameters do not change.

4. The **Inspect** substatement allows verification of computation progress and/or completion of execution within the module.

5. The **Extract** substatement obtains final or intermediate results, as needed, depending on the information returned by the Inspect substatement.

This partitioning allows the mega-program to deal, for instance, with continuously executing modules, as weather simulations, or start multiple executions of functionally redundant modules in parallel and abandon those that fail to make progress. Several substatements are envisaged to manage meta-information [WiederholdWC:92]. These will for instance provide the names, types, and descriptions of parameters available for SET-UP and EXTRACT. The parameters for ESTIMATE and INSPECT are expected to be fixed. ESTIMATEs will include time, cost, and volume of results. They will be reported with a certainty factor, to enable alternatives in scheduling the modules. For some soft real-time environments it will be preferable to have a high-certainty of module performance, even if the estimate of average execution time is higher.

It is important to observe that the implicit parallel operation of CHAIMS should cause the mega-programmer to think naturally of operating in a parallel world, avoiding the current approach where the real-world is described sequentially and subsequent analyses attempt to extract parallelism out of the sequential programs. It is also clear that such a paradigm change cannot be supported in programming languages which merely augment a traditional language with statements that permit parallel operations to be specified over their sequential model.

F.8. Optimization of module configuration.

As indicated above, the use of predefined objects and modules allows rapid composition, but criteria for performance are effectively encapsulated within those objects and modules. Modules can be viewed as large and autonomous objects, and exacerbate benefits and limitations of the object-oriented paradigm. For instance, encapsulation also hides information needed to predict performance. We will develop a scheduler which optimizes the execution sequence for modules based on estimates they provide. The novel concept here is to separate scheduler and modeling of the operations being scheduled [Wiederhold:92o]. Integrated optimization is disabled when schedulers cannot model faithfully the performance of system components because they are remote and encapsulated. In CHAIMS the modules themselves provide their performance estimates via distinct entry points (see Section F.7), enabling the CHAIMS scheduler to re-optimize systems as the configuration and usage parameters change. Parameters to be supplied are principally execution time and result volume, with uncertainties for each [WhangKW:94]. Work of Balzer and Goldman is also relevant here.

The ESTIMATE substatement is a means to obtain necessary information for effective mega-programming, because it avoids the pressure to go over the firewall that encapsulates modules [WiederholdKNSBBS:86]. One of the major barriers to the scalability of object-oriented programming is the need to be able to assess the performance of methods encapsulated in objects when system performance matters. We believe that the CHAIMS substatements are a major contribution to overcoming this barrier. CHAIMS modules can truly follow the precepts of the object-oriented paradigm.

Since CHAIMS execution is expected to take place within a network configuration, further parameters are system rather than module dependent, namely bandwidth and latency of connections. Again, compilers dealing with programming languages do not have the capability to obtain these parameters, and ignore the optimization of module composition. Although a plethora of papers has been written by us and others on optimal module distribution in network configurations [Ahmed:91, SaccaW:84], no tool exists in which such optimization can be performed to the degree that executable code is generated. Within this proposal we only expect to explore existing optimization algorithms for networks, but the framework provided should enable others to test the effectiveness of their ideas.

The result of such a compiled optimization, as illustrated in Fig.3, reduces total bandwidth demands, and also reduces total execution times due to latency reduction gained by bypassing the mega-program for data transfer. The splitting of optimization into (1) estimation of performance parameters by the modules and (2) the CHAIMS scheduler solves a major scalability problem inherent in the concept of encapsulation which is part of the object-oriented paradigm.

F.9 Metrics.

Metrics are needed to establish progress of research and development of technology with respect to performance objectives. Metrics are also desirable for the system projects, although measuring systems is hard, and the final measure of system quality and appropriateness has always been customer acceptance. Relevant metrics should be grounded in customer objectives. The CHAIMS customers are at two levels: the supplier of mega-programs, and the end-user of the mega-programming results.

End-user values are based on having reliable, well-performing, and up-to-date systems. Such end-user values must be translated into metrics that can be technically quantified.

1. Being reliable translates, for instance, into percentage of availability.
2. Performance must at least be adequate, and be comparable with other alternatives.
3. Up-to-date-ness may be measured by the delay from a change request to its implementation.

These measures are affected by each other, and hence must be made in an identical setting. For instance, one can improve performance by careful tuning and rearrangement of modules, but such systems are less likely to be up-to-date. Rapid fixes in turn tend to reduce system reliability, but making systems extremely reliable tends to make them slow and obsolescent. Focusing on a single technology metric can help a specific technology advance, but does not lead to substantial transition into systems [ColemanALO:94].

To be able to compute coherent measures corresponding to the metrics ,a sample collection of modules and a supporting network infrastructure have to be available throughout the research period [HamletT:90]. We will measure our advances through a set of dummy scaffolding modules on a local ethernet configuration. We will measure the ability to keep systems up-to-date by recording the time to replace functionally similar modules and recompiling them into an executing system. We will log and record failures in achieving operation with new, previously untested modules. We will measure pre- and post-performance of modified systems, and report on the effectiveness of optimization strategies over standard module configurations. To reduce the effect of conflicting loads on the network we will collect measures for publication at low-load times.

Metrcs will also be the basis for assessing the module repository.

F.9a Development of a metrics suite (Option 2).
It is best if the same resources we are using to measure our approaches are also available to others, allowing comparison of alternative approaches. Today many software engineering approaches are tested and operated in isolation, and that will be our alternative unless a common test-suite is developed and made available. We include in our proposal an option for the collection and documentation of a set of test modules, but are also willing to adopt system components for composition developed by others.

The modules we would develop for testing should not only model a system of interest to ARPA, but also be broadly understandable [Binder:94]. We suggest the following modules:

1. A module delivering photo images on request.
2. A module containing map information.
3. A module combining the two data types based on named map objects.
4. An output module to show the selected region.

Our internal tests will be based on dummy versions of such modules, with randomized internal performance parameters [Schneidewind:92]. Since we won't be able to control the network, we will measure and report latencies and bandwidth found during our experiments. Successful metrics can still not guarantee acceptance of a system technology. To aid in acceptance, we propose options that empower small innovative companies to develop, introduce, market, and deliver tools based on what we learn in the CHAIMS effort. Such companies will be aided and motivated by metrics work initiated in this proposal [Gelman:94].

Metrics summary.
Measurement is essential to close the feedback loop and drive the systems to relevant closures. Moving to metrics-based systems development goes beyond the use of technology metrics. Relevant metrics need to be defined and justified with respect to the needs of the customer. Support of software engineering researchers and developers by providing and using metrics, requires establishment, access, and maintenance of multiple modules and a network services. An investment in metrics support will pay off in terms of assuring relevance of new technologies to system objectives**.**

F.10 Off-line Glue Code Generation and Invocation (Option 1)
The CHAIMS compiler generates code to invoke the software modules identified in a mega-program. This corresponds to creating off-line the configuration of software modules based on their interface specifications. In some situations there may be a simple direct interface between a calling module and a called module. In other situations, however, the CHAIMS compiler may recognize semantic differences in the two interfaces. The semantic mismatch may be caused by simple conflicts (e.g., protocol and/or notation mismatch), or it may be caused by a conceptual mismatch between the caller and called module (e.g., the order of the arguments, extra arguments, missing arguments, etc.).

The off-line composition of software modules is based the semantic specifications of the module interfaces. As mentioned earlier, the specifications are used by a human programmer to create a mega-program. By making the specifications of modules machine-processable we can ensure that the CHAIMS compiler is more robust and can manage semantic mismatches between the interfaces of modules.

Resolving semantic mismatches involves invocation of glue-code. Glue-code is software which connects functionally independent components. It may be as simple as one or two lines of source code which invokes one module from the next, or it may be more complex, including a series of conversion routines an communication channels which must sit between modules. Glue-code may be generated prior to the execution of an application, or it may be generated on-the-fly at run-time and will be invoked in the data flow among modules.

Option 3 of this proposal builds on the successful work in the Logic Group of Stanford University for on-line software composition. The system, called Agent-Based Software Interoperation (ABSI) [GeneserethSS:94], uses machine-processable software specifications to automatically match a request for a computation to a call to one of more modules that can achieve the computation. In a simple case a request can be matched to a module that can handle it without any change. In other situations the ABSI system may have to perform translation (changing the vocabulary, argument types, argument order), and decomposition (synthesizing a complex computation from a collection of simpler computations that can be executed directly by existing software modules). The ABSI system can provide the foundation for generating glue-code for off-line configuration in the CHAIMS compiler. In the remainder of this section we briefly describe the ABSI system and its application to off-line configuration. Currently the ABSI system relies on on-line configuration. The majority of the effort for Option 3 is to enable the system to support off-line configuration in support of CHAIMS.

As shown in Section I, ABSI provides a tested technology for on-line composition of software modules based on machine-processable specifications. The technology is directly applicable to CHAIMS. The task at hand is to convert the interpreted nature of the current system from on-line configuration to off-line configuration. We know how to compose modules on-line based on their specifications. The task is to generate glue code so that the composition decisions are made once at the beginning at compile-time, rather than making them repeatedly at run-time.

F.11 Technology Investigation with Industry (Option 3).

To enable Technology Transfer we will contract with at least one small software company (let's call it `SSC') to take over advanced development, system integration, introduction, marketing, sales, and long-term service tasks. We believe that such SSCs are the best means for technology transition to application-focused industries. We expect that an SSC will have to contribute substantially, since our experience is that moving from a finished academic prototype to a commercial quality product costs about 5 times the academic development cost [ShortliffePWF:90]. Any system integration costs at an end-user industry, as well as the needed marketing costs, are beyond that estimate. Most of our costs will be documentation and responding to queries from the SSC(s). We expect the SSC(s) in turn will interact with end-users. A university is not able to serve end-users adequately, since student and faculty priorities are different, and oriented towards internal growth. Adding long-term staff for such tasks to University projects is unwise, since research projects are by their very nature cyclical [WiederholdY:94].

A task that is envisaged to fall mainly into the realm of cooperation with industry is the development of a toolbox to serve interoperation. That toolbox will contain components as wrapper generators, filters, format converters and the like, since we cannot expect that modules from distinct sources will be compatible. The selection of appropriate tools will depend on the needs of the initial market. There may be a number of interoperation tools available from the ARPA I3 program and its contractors, as well as from other ARPA and DoD projects that have had to deal with interoperation. Again, a university cannot effectively negotiate for commercial rights, although we envisage that we will be able to get copies of much relevant software for free use within in the university context.

We have experience in dealing with a number of small software companies. Some of them have been started or are staffed by our students or colleagues [OkiPSS:93]. Candidate companies include BCL (Sunnyvale CA), BPG (Newark CA), CALSPAN (Buffalo NY), Crystalliz (Cambridge MA), Cubicon (Burlingame CA), DBStar (San Francisco CA), Interconnect (Menlo Park CA), LXT (Alameda CA), Myriad Software (Palo Alto, CA), Ontek (Laguna Hills CA), Persistence (Sunnyvale CA), PSG (Portland OR), Teknowlegde (Palo Alto CA), Technekron [Berkeley CA), Vitria (Palo Alto CA). This list is, of course, constantly waxing and waning. If this option is funded we will document and announce our technology widely to potential partners, and select one or more. Input from ARPA will be helpful in those negotiation, to assure relevance to DoD. We are of course also willing to work with companies that are funded independently of this contract.

F.12 Summary of the Technical Approach.

The CHAIMS language proposal requires substantial research and experimentation to assure the acceptability of its concepts as well as adequacy of its implementations. For the development of CHAIMS itself we will follow a rapid-prototyping paradigm. The following cycles are proposed; efforts involving partners will be presented as options to give funding flexibility. The extent of overlap of such optional cycles can be varied depending on programmatic needs and capabilities.

Cycle 1. We will define a simple mega-programming language, based on our earlier studies, and validate its consistency and comprehensiveness. Comprehensiveness will be explored by translating the module management aspects shown in examples of other languages which provide modularity into CHAIMS, and if needed, into support modules for glue code. Candidate language examples to be analyzed will be from documentation in PL/1, Ada, Proteus, Rosette (the language used by Carnot), etc.

Cycle 2. Assess the expressiveness of CHAIMS by recoding some existing modularized distributed application into CHAIMS, moving the support code into autonomous modules. Add primitives as needed to the language or as modules in a support library. We will use KQML facilities developed under earlier contracts to allow a rapid initiation.

Cycle 3. Write a compiler for CHAIMS and generate interfacing code for another one of the interfacing protocols, as CORBA. Since CHAIMS is inherently a simple language, such a compiler should be of modest size. On the other hand, it will have to deal with novel primitives and some iterations in development are to be expected. Adapt some currently existing modules from an existing computing system to these standards. Test the compiler using some of these modules, both on single processors and in a distributed setting.

Cycle 4. Document the language, the compiler, and the process, and make it available to other sites and to potential vendors. Verify that it can support software management processes that have been shown to be effective. Assist other sites with their implementations.

Cycle 5. Develop the scheduler for model-independent optimization [Wiederhold:92o]. Augment modules with methods that can export estimates of performance and the certainty of such estimates. Develop a scheduler that takes estimates and rearranges the mega-program execution sequence for optimal execution subject to given constraints. Demonstrate the viability of optimization based on imported estimates.

Cycle 6. Broaden CHAIMS by creating interface code for multiple protocols. This work can overlap prior cycles and the selection of interfaces will be driven by needs of other sites and potential vendors. Provide a repository for CHAIMS-compliant modules.

G: Related Work.

Not surprisingly, there are many other initiatives which address the same problems of concern to this proposal. Fortunately, they tend to be complementary rather than competitive. The need for component technology was already expressed in the 1968 NATO conference [McIlroy:68]. 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++) [Stroustrup:91]. Today many object libraries are available, and they provide a basis for high productivity in writing single-processor programs [Meyer:94]. Experience gained in industry with object-orientation helps in motivating this CHAIMS proposal.

Transitioning of composition to multi-site computing has often been discussed, but solutions have then focused on single protocols. We will profit conceptually from the experience gained in such efforts and practically from the interoperation protocols that have been developed. We have some impressive hand-crafted examples, as the SIMNET and its successors [MillerT:95], tools to rapidly assemble, restructure, and maintain such systems are lacking. However, useful infrastructure standards have been developed. For distributed simulation the DIS protocol provides the basis [IEEE1278:93], for distributed object data configurations the Interface Definition Language (IDL) provides the definition capability [OMG:91], and for knowledge interchange KQML and KIF provide standard conventions [GenesererthK:94]. As indicated in the main body of this proposal, these standards provide the destination code for this CHAIMS proposal. Other initiatives which are relevant to this proposal include:

1. Ada: a language which provides the full range from coding support to composition. Powerful, but complex because of its range, the Ada language does not support distributed computation directly [Libicki:95].

2. Object Libraries: Collections of object class definitions that share a common domain ontology. 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.

3. Automation of Composition: When object libraries are coherent, automatic object composition according to a well-defined architecture becomes feasible [StickelWLPU:94]. Our proposal does not go that far, but relies on the mega-programmer to resolve inconsistencies and even to designate the generation of new modules. Success with CHAIMS can encourage progress in automation as well, because more composable modules may come on the market.

4. Protege.: The software engineering approach established by Musen et al. at Stanford focuses on the reuse of architectural knowledge. Impressive reuse has occurred among

quite different domains [MusenGETP:94]. Such work contributes to the hyper-programming concepts presented in this BAA, and can help the mega-programmer either formally or informally.

5. SimQL: The SimQL project proposes to define an interface language to access simulations. The results of SimQL research will merge well with this proposal, since simulations are a very important instance of large-scale modules, often requiring dedicated computation resources. It is receiving initial funding via NIST.

6. Architecture Description Languages (ADLs): ADLs provide a conceptual and general definition of an architecture type, often in graphical form. As indicated above, a CHAIMS mega-program describes the instance of the architecture of the application system. Since CHAIMS is primarily intended for execution, 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.

7. Interoperation Standards: Real or potential standards for interoperation, as OLE, CORBA, KQML, and DCE were introduced above as the machine-language for CHAIMS. We depend critically on them, and hope they will succeed in making Server-client middleware an effective development concept. Their weakness is generally that they operate at a low level. Furthermore, their very (competitive) nature does not favor general interoperation, nor support optimization. While it is accepted that distributed heterogeneous systems will be common, few operational examples exist.

8. Facilitation: The Stanford Logic Group has developed *facilitators* which use the declarative module specifications to automatically coordinate their interoperation. The architecture supports anonymous interoperation, where a request for a computation need not identify the modules to call (facilitators can automatically select the appropriate module(s) to service the request). The distinctive feature of this approach is the expressiveness of the declarative specification language, which enables sophisticated support for computation requests, e.g., decomposing complex requests into a collection of simpler requests, and translating between the interface of a requesting module and the interface of a module that can service the request. In option 1 we will exploit this technology to generate the glue code that is needed by CHAIMS to link modules that have semantic mismatch.

H: Key personnel.

Gio Wiederhold is a Professor of Computer Science at Stanford University and has courtesy faculty appointments in Medicine and Electrical Engineering. He works on applying knowledge-based techniques to database management, information systems, security, and software construction and maintenance.

He will put 50% effort on this proposal throughout the period. He currently receives research funding from Commercenet (an ARPA TRP) 50% and TIHI (Trusted Inter-operation of Health Care Information) an SRI NSF subcontract. He expects partial support for SiMQL (A Simulation Query language to integrate simulations as modules into larger programs) from NIST and for Mediation in Training from ARPA. Some of these efforts will complete before CHAIMS will be funded, and he can reduce his effort on Commercenet and turn them over to a research associate.

Gio is a member of the Air Force Scientific Advisory Board, and of the Technology Advisory Board for Advanced Digital Simulation Systems at the Institute of Defense Analysis. He is a member of the ACM publications board, focusing on the move towards electronic dissemination of information. He is an advisory member of the ARPA-Rome Laboratory Planning Initiative (ARPI), a member of the Military Operations Research Society (MORS), and a member of the International Federation of Information Processing Societies (IFIP) Working Group WG2.6 for electronic data management. He is also an associate editor of several journals. Wiederhold has published 4 books and more than 250 publications in computing and medicine, including a widely used McGraw-Hill textbook, DATABASE DESIGN. He has been chairman and program chairman of several conferences.

At Stanford Gio Wiederhold is a member of the Computer Systems Laboratory, the Medical Information Science Program, and the Centers for Integrated Systems (Dep. of Elec. Eng.), Facility Engineering (Dep. of Civil Eng.) and the Stanford Computer Industry Project (SCIP) of its Business School. Past work on the application and development of knowledge-based techniques to database management, information systems, and software construction and maintenance, initiated with the KBMS project at Stanford in 1977, has now become an active research field in its own right. Results derived from this work are found in general system design, medicine, information systems, and planning applications. He has supervised over 25 PhD students. Current research projects address Heterogeneity of Information within Commercenet and Trusted Interoperation of Healthcare Information. He currently teaches an experimental course on `Large-Scale Software Construction' (CS446).

From 1991 to 1994 he was the Program Manager for Knowledge-Based Systems at ARPA, initiating programs in Intelligent Integration of Information (I3) and in Persistent-Object Bases (POB). He also participated in establishing directions for a Software Foundations program, specifically in composition technology for large, multi-site systems.

Professor Wiederhold received a degree in aeronautical engineering in Holland in 1957 and shortly after came to the United States. After gaining 16 years of industrial experience, he returned to school and earned a PhD in 1976 in Medical Information

Science from the University of California in San Francisco. He has been on the Stanford faculty since that time. He is a fellow of the ACMI, the IEEE, and the ACM, as well as a member of the AAAS, AAMI, AIAA, and AAAI.

He has consulted for many governmental and commercial enterprises, among them the United Nations Development Programme, the US Department of Health and Human Services, various US defense agencies, and, of course, with Silicon Valley innovators.

Professor Wiederhold's address is Department of Computer Science, Stanford University, Stanford, CA 94305-2140, or Gio@cs.stanford.edu.
His home page is at http://www-db.stanford.edu and includes a full vita.

Narinder Singh is a Research Associate in the Computer Science Department at Stanford University. Narinder received his Ph.D. in 1985 from Stanford. His thesis focused on the problem of representing and reasoning about complex discrete devices (e.g., simulating, testing, diagnosing digital computers). Narinder published a book based on his thesis on the application of automated reasoning to test generation for digital devices.

Dr. Singh will participate at a 25% level on this proposal. He now receives most of his support from Commercenet and a subcontract of an ARPA-funded TRP on Composition. He will be able to free up the time needed for participation in CHAIMS.

Dr. Singh's address is Department of Computer Science, Stanford University,
Stanford, CA 94305-2140, or singh@cs.stanford.edu

Michael Genesereth (the leader of the proposed Option 1 effort) is an Associate Professor of Computer Science at Stanford University, and is director of the Center for Information Technology.

He will spent 10% effort on direction of Option 1, if it is funded. He is now the Principal Investigator of Stanford's efforts on CommerceNet and of an ARPA-funded I3 project on facilitators and interoperability. He is also PI of Stanford's efforts on an ATP project on Software Composition that has been selected for funding by NIST. He received his Ph.D. in Applied Mathematics from Harvard in 1978. Prof. Genesereth is most known for his work on logical systems and applications of that work in engineering automation and software interoperation. He is the author of a popular textbook in Artificial Intelligence, was program chairman for the national AI conference, and serves on the editorial board of the Artificial Intelligence Journal. He is a member of the advisory board for the ARPA Knowledge Sharing Effort and co-chairman of the Interlingua Committee.

I: Previous Accomplishments.

Prof. **Gio Wiederhold** has a long history in innovative computing software systems, both prior to his entering academia and subsequently. Directly relevant to this proposal are four achievements.

1. In 1965 Gio initiated the development of a research support system (Advanced Computer for Medical Experimentation, ACME) [SandersW:67], which included distributed computation, real-time data acquisition, and provisions for modular interfaces [WiederholdB:70]. The Time-Oriented Database System (TODS) within ACME provided a relational model, to help users understand its functions, but internally mapped data access to a transposed representation for high speed performance in medical data analyses [Wiederhold:66]. This system provided an early example of hiding complex data structures into simple, clear external concepts, greatly enhancing its usability. The system is still in regular use, maintained and marketed by a small software company, IAC of Tahoe City CA as MedLog [WiederholdFW:75]. The architecture has been adopted by the statistical services of Great Britain and Canada.
2. In 1985/86, Gio participated in the definition of an effort to upgrade the World-Wide Military Command and Control System (WWMCCS) with more modern technology [WiederholdBHKNSS:86]. In the process an Ada-based database system was designed and partially implemented. This system exploited Ada facilities for modularization [WiederholdKNSBBS:86]. An important facet of the architecture was interface control, with a view towards providing security through its architecture, while most modules would not require full validation [SpoonerKWSH:86].
3. Starting in 1989, Gio Wiederhold, Arthur Keller, and others developed with ARPA sponsorship concepts of mediating modules, which break the synchronicity constraint that causes much of the high cost of software maintenance [Wiederhold:92m]. Within the mediator paradigm we have worked on approaches that should lead to an approximately linear growth of cost of maintenance with the growth of software systems, whereas the current cost curve is approximately quadratic [Wiederhold:95s]. We are now participating in an ARPA-sponsored effort to quantify these metrics, and are developing a good sense of various waypoints. At the outer extreme are systems as WWMCCS, where maintenance is being continuously deferred because of high cost of maintenance, even though the design is modular and uses a pre-cursor of client-server architectures [WiederholdBHKNSS:86].
4. Experience with the U.S. Air Force VHSIC program showed weaknesses in the then emerging object-oriented paradigm for large-scale applications [WiederholdSK:85]. Desired improvements in the architecture were defined [WiederholdWN:89]. Subsequently, Gio and several of his students formulated and implemented a concept of *View-Objects*, which enables object-oriented access to relational databases [BarsalouW:90]. This technology is now central to the products of Persistence Corporation and has become part of the SunSOFT Distributed Objects Environment [KellerJA:93].

Before joining the Stanford faculty Gio led the development of flexible real-time data-acquisition, file and database systems to support clinical research at Stanford. Recent research into workstations for design and genomic experiment planning integrates these interests. Prior to Stanford major software projects included systems to predict the performance of solid-rocket fuel, the optimization of oil refining columns, maintenance of the IBM FORTRAN II and III compilers, analysis of hybrid corn statistics, analysis of rent-versus-purchase of IBM's mainframe component inventory (much smaller in 1960 than now), the partitioning of a large-scale insurance risk re-distribution process (CSO), and the development of early information retrieval systems, using touch-tone data entry and voice answer-back.

The results of these research efforts resulted in a clear, high level model of distributed computation [WiederholdWHCSCQWDR:89]. For distribution, a model describing data constraints affecting asynchronous operation of systems was developed [WiederholdQ:87].

Dr. **Narinder Singh** was centrally involved in the Helios design assistant, where he demonstrated the utility of reasoning with declarative design models in a computer-aided-design environment. Following this he helped develop the DesignWorld system, which supported the entire life-cycle of small-scale digital circuits, from design to assembly and repair. A key to realizing the DesignWorld system was the development of software technology to support the integration of legacy (and custom-built) CAD tools running on different machines, different operating systems, different interoperation technologies, and implemented in different languages.

Currently Narinder is coordinating the ARPA-funded Agent-Based Software Interoperation project. The goal of the research is to develop a system that simplifies the task of software developers by placing much of the burden of interoperation on the system. The software modules being integrated include legacy programs, data-bases, knowledge bases, and custom software. The interoperation is based on machine-processable specifications, which the system uses to integrate modules and to resolve semantic mismatches between interfaces (performing translation, problem decomposition, information integration, etc.) This research was originally done in collaboration with Hewlett Packard, and it is currently being used in the CommerceNet project in support of integrating business information from a variety of sources.

Prof. **Mike Genesereth** directs work in the Stanford Logic Group of direct relevance to this proposal includes research on Agent-Based Software Interoperation (ABSI). The ABSI system has been demonstrated to work with legacy modules (e.g., an Excel spreadsheet, student databases) running on different platforms (e.g., 486 PC compatibles, Sun workstations, Apple Macintoshes), running a variety of operating systems (Windows NT, Solaris, IRIX, Macintosh System 7). There is an API for C, C++, and Lisp. It hides all the details associated with network communication from the programmer. Each API uses TCP/IP and SMTP to communicate with the facilitator (some also support OLE, and

AppleEvents), but changing the communication protocol (e.g., to CORBA) is straightforward.

ABSI has been used in a collection of interoperation experiments, including an integrated design, manufacture, and diagnosis system for digital circuits [Genesereth:91], a multi-domain simulation of a robotic arm, and integrated CAD tools for civil engineering [Cutkosky:93]. It is currently being used in the CommerceNet project [Keller:95], which provides smart search for product information using heterogeneous on-line catalogs, ordering, billing, etc. It is also currently being used in the Infomaster project, where a collection of hybrid and distributed information sources are integrated (automatically parsed newspaper web pages, data bases, etc.), and a user is presented with an automatically generated WWW front-end to pose queries to the system (e.g., Netscape).

The ABSI system relies on a novel specification-sharing approach to software interoperation. The services of the system are provided by *facilitators*, which are responsible for all coordination. Software modules provide machine processable descriptions of their capabilities. Modules can be realized in different programming languages, and they can run in different processes on different machines. In addition, modules can be dynamic -- at run time modules can join the system or leave.

J: Facilities.

The Computer Science Department at Stanford University provides excellent general facilities for experimental and theoretical research. The department supports a world-class Computer Science library, including various computerized search systems. The department operates an infrastructure communication network, serving small and large computer systems for research, education, and administration. These systems are connected to SUNet, the university-wide Ethernet system, to the nation-wide NSF and BARRNet research networks, and to the world-wide Internet. By the time this proposal is awarded, the department will have moved to the Gates Computing Science Building, where both 10T and 100T connections will be available.

Within the Computer Science Department, the Stanford Database Group runs a cluster with approximately 20 workstations, primarily DEC 5000's and IBM RS/6000's. The cluster is serviced by the Computer Science Department Computer Facilities staff, which maintains an up-to-date software environment, including a Sybase DBMS, an Oracle DBMS, and plans installation of one or more Object-Oriented DBMS's.

We request only two pieces of equipment for the database group within this proposal. One will be the UNIX workstation which will host the mega-programming language, its compiler, and its operation. The second will be a networked large PC, probably running Windows NT, in order to have access to a module platform which is becoming increasingly prevalent. DCE and OLE will both be available on the planned PC system. We will also have to obtain software that complies with, and supports, the interoperation standards we will exploit. Other modules to be invoked by the mega-programs will be placed on existing equipment.

The Logic Group of the Computer Science Department at Stanford University has a collection of about eight Unix workstations. These are primarily Sun Sparc 4 machines. We also have a collection of desktop and portable Macintoshes, and one IBM PC compatible. An additional portable as well as funds to update earlier equipment are requested. There is one Sybase data-base server running on a NeXT workstation. The infrastructure supporting the Logic Group computational facilities is provided by the Computer Science Computer Facilities group, and it is identical to that for the Database Group.

Stanford Universities Administration is in the process of updating its infrastructure computer services, based on a distributed computing model under OSF DCE [Lockhart:93]. To the extent that we do not interfere with University operations, we will interact and profit from these resources. Gio Wiederhold is a member of the external advisory committee to Stanford's Administrative Computing organization.

K: Cost Breakdown.

	7/1/96-6/30/97	7/1/97-6/30/98	7/1/98-6/30/99	9/1/96-8/31/99
Salaries				
Gio Wiederhold, 50%	51,000	53,040	55,162	159,202
Narinder Singh, 25%	18,200	18,928	19,685	56,813
Postdoc, 50%	20,000	20,800	21,632	62,432
Marianne Siroker, Admin,10%	4,056	4,218	4,387	12,661
Student research assistants	(2.5)	(3.5)		
50% AY, 100% summer	69,829	72,622	75,527	217,978
Subtotal Salaries	163,085	169,608	176,393	509,086
Staff Benefits @ 26.2% FY96,97 25.5% thereafter	42,728	32,290	25,721	100,739
Travel-domestic	6,000	6,240	6,490	18,730
Computer facilities	4,800	7,300	7,592	19,692
Expendable Materials (incl. software)	4,000	4,160	4,326	12,486
Subtotal, Direct Costs	**220,613**	**219,598**	**220,522**	**660,733**
Indirect Costs @ 61%	134,574	133,955	134,518	403,047
Capital Equipment	37,000		6,000	43,000
Tuition		28,143	32.412	60,555
Total Budget	**392,187**	**381,696**	**393,452**	**1,167,335**

Preliminary Budget for Options

Option 1: Glue-code *1/1/97 - 1/1/2000*

Salaries and Benefits (50% Research Associate 50% Programmer, Student)	$116,000
Other	4,000
Indirect Costs	72,000
Equipment	12,000
Total:	**$204,000/year**

Option 2: Metrics suite *7/1/97 - 12/31/97*

Salaries and Benefits (10% Gio Wiederhold 40% Postdoc, 2 students)	$ 57,000
Other	9,000
Indirect Costs	40,000
Equipment	10,000
Tuition	18,000
Total:	**$134,000** (one year only)

Option 3: Transition *7/1/97 - 6/30/2000*

Salaries and Benefits (20% Gio Wiederhold 40% Postdoc)	$ 48,000
Other	9,000
Indirect Costs	35,000
Subcontract	300,000
Total:	**$392,000/year**

K.1 Budget justification.

Salaries
Escalation factors are 4% each year for cost of living increases.

Travel
We have budgeted 3 East Coast trips each year at $2,000/trip

5 days per diem @ $150/day	$ 750		
round trip airfare (coach)	1000		
misc. expenses	250	total per trip	$2,000

Computer facilities
Cost is estimated at $4800 per year for the first year for

network charges: $30 per person per month:	$ 960
printing: $0.10 per page:	$ 1200
maintenance:	$ 840
systems administration $150 per month	$ 1800

The budget for the 2nd and 3rd year includes an extra $2500/year for access to distributed computing systems.

Expendable Materials
Telephone expense based on $600/year; Photocopy expense based on $600/year
Software expense based on $2500/year; Supplies expense based on $300/year

Equipment
We propose to buy one PC ($10,000) and two UNIX workstations ($13,500) in the first year. In the third year we expect to need an upgrade for large-scale prototyping ($6,000). These machines will be used to implement CHAIMS, house the KQML specification repositories, and develop the CHAIMS/KQML APIs. In addition, in we plan to use the three machines together to develop and evaluate the system in a distributed setting, taking into account the cost metrics of the network and module specifications.

Tuition
Beginning with FY98, the 1993 revisions to OMB Circular A-21 require the University to directly charge tuition. Therefore, we have budgeted for tuition for the 3.5 student research assistants during the 2nd and 3rd year. Tuition for a 50% time research assistant is 62% of the full tuition rate. 40% of the full tuition rate is included as a line item. Stanford will cost share the remaining 22%. Tuition is exempt from indirect costs. Effective 10/1/97 the staff benefits rate is not applied to the research assistant salary, since staff benefits are not charged on student salaries when tuition is direct charged.

L: References.

A more complete bibliography and annotations can be found under http://db.stanford.edu/pub/gio/CS446/biblio.html.

[AdlerW:92] P.S.Adler and Terry Winograd: *Usabililty; Turning Technology into Tools*; Oxford Un. Press, 1992.

[AgarwalKSW:95] Shailesh Agarwal, A.M. Keller, K. Saraswat, and G. Wiederhold: “Flexible Relation: An Approach for Integrating Data from Multiple, Possibly Inconsistent Databases" ; *Proc. IEEE Data Engineering Conference*, Taipei, Taiwan, March 1995.

[Ahmed:91] R. Ahmed et al.: “The Pegasus Heterogeneous Multidatabase System”; IEEE Computer. Dec.1991, Vol.24 No.12.

[Andrews:93] Tim Andrews: "Designing Linguistic Interfaces to an Object Database or What do C++, SQL, and Hell have in Common?"; in *Database Programming* (DBPL-4), edited by C. Beeri, A. Ohori, and D. E. Shasha, Springer-Verlag, 1993, pp. 3-8.

[Arnold:93] Robert S. Arnold: *Software Reengineering*; IEEE Press, 1993.

[Bach:94] James Bach: *Process Evolution in a Mad World*; Borland International, Scotts Valley CA, 1993.

[Barnhart:94] Andy Barnhart: "Component-based solutions with OLE 2.0"; *Software Development*, Vol.2 No.9, Sept. 1994,page 47-51.

[BarrT:95] Avron Barr and Shirley Tessler: *Software Industry Study*; Stanford Computer Industry Project, 1995, http://scip.stanford.edu/scip.

[BarsalouW:90] Thierry Barsalou and Gio Wiederhold: "Complex Objects For Relational Databases"; Computer Aided Design, Vol. 22 No.8, Buttersworth, Great Britain, October 1990, pages 458-468.

[Basili:90] Victor Basili: "Viewing Maintenance as Reuse-Oriented Software Development"; *IEEE Software*, Vol.7 No.1, Jan. 1990, pp.19-25.

[Binder:94] Robert Binder: “Object-Oriented Software Testing”; special topic issue, *Comm.ACM*, Vol.37 No.9, Sept. 1994.

[Blum:92] Bruce Blum: *Software Engineering, A Holistic Approach*; Oxford Un. Press, 1992.

[Boehm:84] Barry W. Boehm: “Software Engineering Economics"; *IEEE Trans. Software Eng.,* Vol.10 No.1, Jan.1984, pp.4--21.

[BoehmS:92] Barry W. Boehm and William Scherlis: “Mega Programming (preliminary version)"; *Proceedings DARPA Software Technology Conference 1992*, April 1992, Meridien Corp., Arlington VA, pages 63-82.

[Brockschmidt:94] Kraig Brockschmidt: Inside OLE 2; Microsoft Press, 1994, 977pp.

[Charette:79] R.H. Charette: *Software Engineering and Risk Analysis Management*; McGraw-Hill 1979.

[ColemanALO:94] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman: "Using Metrics to Evaluate Software Systems Maintainability"; *IEEE Computer*, Vol.27 No.8, Aug.1994, pp.44-49.

[CourtneyJB:93] A. Courtney, W. Janssen, D. Severson, M. Spreitzer, and F.Wymore: Inter-Language Unification, rel.1.5; Xerox PARC, ISTL-CSA-94-01-01, No.P94-58, May 1994, ftp://ftp.parc.xerox.com/pub/ilu/ilu.html.

[Cutkosky:93] M.Cutkosky et. al.: "PACT: An Experimental in Integrating Engineering Systems"; *IEEE Computer*, vol.26 No.1, 1993, pp. 28-37.

[DelisR:92] A. Delis and N. Roussopoulos: "Performance and Scalability of Client-Server Database Architectures"; *18th Int. Conf. on Very Large Data Bases*, Vancouver, British Columbia, Canada, 1992, 610-623.

[Dertouzos:86] Michael L. Dertouzos: "The Multiprocessor Revolution: Harnessing Computer Together"; *Technology Review*, Vol.89 No.2, March 1986, MIT, pages 44-57.

[ECMA:93] ECMA TC33: *Portable Common Tools Environment* (PCTE), 1992.

[FininWa:94] Tim Finin, Gio Wiederhold, et al ; available from the WWW with the URL http://www.cs.umbc.edu/kqml/kqmlspec/spec.ps

[FriederS:91] Ophir Frieder and Mark E. Segal: "On Dynamically Updating a Computer Program: from Concept to Prototype"; J. of Systems Software, Elseviers, Vol.14, pp.111-128, 1991.

[GarlanA:95] David Garlan, Oscar Nierstrasz, Theo Dirk Meijler, Gio Wiederhold, Frank Manola, Sandra Heiler, an Jeff Sutherland: "Research Directions in Software Engineering"; *ACM Comp. Surveys*, Vol.27 No.2, June 1995, pages 256-276.

[Gelman:94] Stacey Gelman: "SILVER BULLET: An Iterative Model for Process Definition and Improvement"; AT&T Technical Journal, Vol.73 No. 4, Jul-Aug.1994, pp.35--45.

[GeneserethK:94] M.Genesereth and S.Ketchpel: "Software Agents"; Comm.ACM, Vol.37 No.7, July 1994.

[Genesereth:91] M. Genesereth: "DesignWorld"; in the *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, Sacramento, CA, April 1991, pp. 2785-2788.

[GeneserethSS:94] Genesereth, M., Singh, N. and Syed, M. "A Distributed and Anonymous Knowledge Sharing Approach to Software Interoperation," in the Proceedings of the International Symposium on Fifth Generation Computing Systems, 1994, pp. 125-139.

[Gibbs:94] W.Wayt Gibbs: "The Software Crisis"; Scientific American, Sep.1994.

[HamletT:90] D. Hamlet and R. Taylor: "Partition Testing does not inspire Confidence"; *IEEE Trans. on SW Engineering*, Vol.SE-16, Dec.1990, pp. 1402-1411.

[IEEE1278:93] IEEE Standard*: Protocols for Distributed Simulation Applications: Entity Information and Interaction*; IEEE Standard 1278, IEEE Computer Society 1993.

[Jackson:82] M.A. Jackson: System Development; Prentice Hall, 1982.

[Kapoor:91] M.Kapoor: "A Software Design Manifesto: Time for a Change"; *Dr. Dobbs Journal*, Jan.1991.

[Karr:93] Robert N. Karr: *Data Management Issues Associated with Stovepipe Systems*; U.S General Services Administration, Information Resources Service, KMP-94-1-1, Oct. 1993.

[Keller:95] A. Keller: "Smart Catalogs and Virtual Catalogs";in *USENIX Workshop on Electronic Commerce*, August, 1995.

[KellerJA:93] A.M. Keller, R. Jensen, and S. Agarwal: "Persistence Software: Bridging Object-Oriented Programming"; and Relational Databases. {\em ACM SIGMOD, International Conference on Management of Data}, May 1993.

[Kruger:92] Charles Kruger: "Software Reuse"; ACM Comp. Surveys, vol.24 No.2, June 1992.

[Libicki:95] Martin C. Libicki: *Information Technology Standards*; Digital Press, 1995.

[Lockhart:95] H.W. Lockhart, jr.: *OSF DCE, A Guide to Developing Distributed Applications;* McGraw-Hill, 1995.

[LowryM:91] Michael R. Lowry and Robert D. McCartney: Automating Software Design; AAAI press/ MIT Press 1991.

[Luqi:90] Luqi: "A Graph Model for Software Evolution"; *IEEE Trans. on SW Eng.,* Vol. SE16 No.8, 1990, pp.917--927.

[MarcoB:87] A.Marco and J.Buxton: *The Craft of Software Engineering*; Addison-Wesley, 1987.

[McIlroy:68] Doug McIlroy: "Mass-produced Software Components"; in Buxton et al.: *Software Engineering Concepts and Techniques*; Van Norstrand Reinhold, 1976.

[MerksDRDC:92] E.A.T. Merks, J.M. Dyk, and B.G. Ryder: Language Design for Program manipulation"; *IEEE Trans.SW Eng*., vol 18 No.1, Jan.1992, pp.19-32.

[Meyer:94] Bertrard Meyer: *Reusable Software: The Base Object-Oriented Component Libraries*; Prentice-Hall, 1994.

[MillerT:95] Duncan C. Miller and Jack A. Thorpe: "SIMNET: The Advent of Computer Networking"; *Proceedings of the IEEE*, August 1995, vol.83 No.8, pages 1116-1123.

[Mills:94] Elinor Mills: "Users Fear Hassles In Using Component Software"; *InfoWorld*, Vol.16 No.38, Sept 19, 1994, pages19-20.

[MusenGETP:94] M.A. Musen, J. H. Gennari, H. Eriksson, S. W. Tu, and A.R. Puerta: *PROTEGE II: Computer Support For Development of Intelligent Systems From Libraries Of Components*; Knowledge Systems Laboratory, Medical Computer Science, Technical Report KSL-94-60.

[NechesFFGPSS:93] R. Neches, R. Fikes, T. Finin, T.R. Gruber, R. Patil, T. Senator, and W.R. Swartout: "Enabling Technology for Knowledge Sharing"; *AI Magazine*, Vol.12 No.3, pages:37--56, 1993.

[Newport:93] J.P. Newport: "The Growing Gap in Software"; *Fortune*, vol.113, Apr.1992, pp.132-142 .

[OkiPSS:93] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen: "The Information Bus -- an Architecture for Extensible Distributed Systems*; 12th Symp. on Operating Systems Principles* (SOSP12), pp.58-68, DEC 1993.

[OMG:91] Object Management Group*: The Common Object Request Broker: Architecture and Specification*; OMG Document 91.12.1, OMG and X/Open, distributed by QED-Wiley, Wellesley MA, 1991.

[OSF:91] Open Systems Foundation: *Distributed Computing Environment* (DCE); http://www.OSF.org, 1991.

[Parnas:79] D.L. Parnas: "Designing Software for Ease of Extension and Contraction"; *IEEE TSE*, Vol.5 No.2, March 1979, pp.128-138.

[Parnas:95] David Lorge Parnas: "Fighting Complexity"; *IEEE TC ECCS Newsletter*, Vol.2 No.2, October 1995.

[Rettig:91] Marc Rettig: "Testing made Palatable"; *Comm.ACM*, Vol.34 No.5, May 1991, pp.25-29.

[RosenberryKF:94] Rosenberry, Kenney, and Fisher: *Understanding DCE*, OReilley 1994.

[SaccaW:84] Domenico Sacca and Gio Wiederhold : "Database Partitioning in a Cluster of Processors"; *ACM TODS*, Vol.10 No.1, Mar. 1984, pages 29-56.

[SamadzadehM:95] Mansur Samadzadeh and Mansour Zaid: *Proc. ACM SIGSoft Symposium on Software Reusability*; ACM, April 1995.

[Schneidewind:87] N.F. Schneidewind: "The State of Software Maintenance"; *IEEE Trans.on Software Engineering*, Vol.SE13 Mar,1987, pp.303-310.

[Schneidewind:92] N.F. Schneidewind: "Methodology for Validating Software Metrics"; *IEEE Trans. on Software Engineering*, Vol.SE18, May 1992, pp.410-422.

[ShortliffePWF:90] Edward Shortliffe, Leslie Perrault, Gio Wiederhold, and Larry Fagan (eds): *Medical Informatics: Computer Applications in Health Care*; Addison-Wesley, April 1990, 715 pages.

[Smith:90] Smith, D.R., KIDS: A Semi-Automated Program Development System; *IEEE Transactions on Software Engineering*, Vol.16 No.9, Special Issue on Formal Methods, September 1990, 1024-1043.

[SandersWA:67] William J. Sanders, Gio Wiederhold, et al: "An Advanced Computer System for Medical Research"; *Proceedings of the AFIPS Conference*, Vol. 31, Washington, DC, Thompson Books, 1967, pages 497-508.

[SpoonerKWSH:86] David Spooner, Arthur M. Keller, Gio Wiederhold, John Salasin, and Deborah Heystek: "Framework for the Security Component of an Ada DBMS"; *Proc. Twelfth International Conference on Very Large Data Bases* (VLDB), Kyoto, Japan, August 1986, Morgan Kaufmann pubs., pages 347 to 354; also in *Proceedings of the*

Database Security Workshop, National Computer Security Center, Fort Meade MD, June 1986.

[StickelWLPU:94] Mark Stickel, Richard Waldinger, Michael Lowry, Thomas Pressburger, and Ian Underwood: "Deductive Composition of Astronomical Software from Subroutine Libraries"; *Twelfth Intern.,Conf. on Automated Deduction*, Nancy, France, Jun 1994, pages 341-355.

[Stoyenko:95] Alex Stoyenko: "Engineering of Complex Computer Systems -- A New Challenge for Computer Types Everywhere"; *IEEE TC ECCS Newsletter*, Vol.2 No.2, October 1995.

[Stroustrup:91] Bjarne Stroustrup: *The C++ Programming Language*, 2nd ed; Addison Wesley, 1991.

[Swartz:94] Jon Swartz: "OpenDoc, OLE squaring off"; *MacWEEK* Vol.8 No.37, 19 Sept 1994 page 1-2.

[TomlinsonCMW:93] Christine Tomlinson, Phil Cannata, Greg Meredith, and Darrell Woelk: "The extensible Services Switch in CARNOT"; *IEEE Parallel & Distributed Technology*; Vol.1 No.2, May 1993, pp16-20.

[Tracz:95] Will Tracz: *Confessions of a Used Program Salesman*; Addison-Wesley 1995.

[Udell:94] John Udell: "Component Ware"; *Byte*, May 1994, pp.46-56.

[Wegner:95] Peter Wegner: "Interactive Foundations of Object-Based Programming"; IEEE Computer, Vol.28 No.10, Oct.1995, pp.70-72.

[WhangKW:94] Kyu-Young Whang, Sang-Wook Kim, and Gio Wiederhold: "Dynamic Maintenance of Data Distribution for Selectivity Estimation"; *VLDB Journal*, Vol.3 No.1, Jan.1994, pages 29-51.

[Wiederhold:66] Gio Wiederhold: "A Summary of the ACME System"; *Proceedings of the ONR Computer and Psychobiology Conference*, Monterey, CA, Office of Naval Research, May 1966.

[Wiederhold:92m] Gio Wiederhold: "Mediators in the Architecture of Future Information Systems"; *IEEE Computer*, March 1992, pages 38-49.

[Wiederhold:92o] Gio Wiederhold: "Model-free Optimization"; *Proceedings DARPA Software Technology Conference 1992*, April 1992, Meridien Corp., Arlington VA, pages 83-96.

[Wiederhold:93] Gio Wiederhold: "The Role of Government in Standards"; *ACM Standards Bearer*, Dec.1993, pages 2-6.

[Wiederhold:95] Gio Wiederhold: *Mediation and Software Maintenance*; Stanford Computer Science Department Technical Note STAN-CS-TN-95-26, October 1995.

[WiederholdB:70] Gio Wiederhold and Gary Breitbard: "A Method for Increasing the Modularity of Large Systems"; IEEE Computer, March-April, 1970, Vol.3 No.2.

[WiederholdBHKNSS:86] Gio Wiederhold (chairman), Murray Berkowitz, Deborah Heystek, Arthur M. Keller, Shamkant Navathe, David Spooner, and John Salasin: *Ada Foundation*

Technology, Volume V: Software Requirements for WIS Database Management Prototypes, Institute for Defense Analysis, Paper P-1893, December 1986.

[WiederholdFW:75] Gio Wiederhold, James F. Fries, and Stephen Weyl: "Structured Organization of Clinical Databases"; Proceedings of the National Computer Conference, 1975, AFIPS Vol.44, pages 479-485.

[WiederholdKNSBBS:86] Gio Wiederhold, Arthur M. Keller, Sham Navathe, David Spooner, Murray Berkowitz, Bill Brykczynski, and John Salasin: "Modularization of an Ada Database System"; *Proc. International Pre-VLDB Symposium*, Beijing PRC; also, in S.B. Yao, S-H. Sa, Z-S. Zhong(eds), China Assoc.for Science and Technology, Aug. 1986, pages 202-232; also, extended abstract in *Proc.Sixth Advance Database Symposium*, Information Processing Society of Japan, Aug. 1986, pages 135-142.

[WiederholdQ:87] G. Wiederhold and X-L. Qian: “Modeling Asynchrony in Distributed Databases"; *IEEE Data Engineering Conference* 3, Los Angeles, Feb. 1987.

[WiederholdSK:85] Gio Wiederhold, John Smith, and Phillip Kaufman: "Models for Engineering Information Systems"; abstract in Proceedings of the 1985 VHSIC Conference, Silver Spring, Maryland, December 1985.

[WiederholdWN:89] Gio Wiederhold, Marianne Winslett, and Lt. Nicholas Naclerio: "Layering an Engineering Information System"; Digest of Papers, IEEE CS Spring COMPCON 34, February 1989, pages 444-449.

[WiederholdWC:92] Gio Wiederhold, Peter Wegner, and Stefano Ceri: "Towards Megaprogramming"; *Comm.ACM*, Vol.35 No.11, November 1992, pages 89-99.

[WiederholdWHCSCQWDR:89] G.CM Wiederhold, M.G. Walker, W. Hasan, S. Chaudhuri, A. Swami, S.K. Cha, X-L. Qian, M. Winslett, L. DeMichiel, and P.K. Rathmann: “KSYS: An Architecture for Integrating Databases and Knowledge Bases"; in Amar Gupta (ed.) *Heterogeneous Integrated Information Systems*, MIT, 1989.

[WiederholdY:95] Gio Wiederhold and Yelena Yesha: *Dealing with Asynchrony in Technology Transfer*; abstract for a presentation given at NIST, March 1995; available on http://www-db.stanford.edu/pub/gio/1995/TT.html .

[ZelkovitzA:84] M.V. Zelkovitz et al: "Software Engineering Practices in the US and Japan; *IEEE Computer*, June 1984, pp 57-66.

---------------------- o ---------------------- o ----------------------- o ------------------------

Attachments (provided with the first 4 copies):

1. Gio Wiederhold, Peter Wegner, and Stefano Ceri: "Towards Megaprogramming"; *Comm. ACM*, Vol.35 No.11, November 1992, pages 89-99.
2. Gio Wiederhold: *Mediation and Software Maintenance*; Stanford Computer Science Department Technical Note STAN-CS-TN-95-26, October 1995.
3. Gio Wiederhold: "Model-free Optimization"; *Proc. DARPA Software Technology Conference 1992*, April 1992, Meridien Corp., Arlington VA, pages 83-96.