Towards Megaprogramming:
A Paradigm for Component-Based Programming

This paper is based on a paper published in the

Communications of the ACM, Vol.35 No.11, November 1992, pages 89-99.

Gio Wiederhold [1], Peter Wegner [2], and Stefano Ceri [3]

January 14, 1996

Abstract

Megaprogramming is a technology for programming with large modules called *megamodules* that capture the functionality of services provided by large organizations like banks, airline reservation systems, and city transportation systems. Megamodules are internally homogeneous, independently maintained software systems managed by a community with its own terminology, goals, knowledge, and programming traditions. Each megamodule describes its externally accessible data structures and operations and has an internally consistent behavior. The concepts, terminology, and interpretation paradigm of a megamodule is called its *ontology*.

The objective of this paper is to provide a conceptual framework for the study and development of megaprogramming and to identify technical problems to be addressed; we do so by proposing a *megaprogramming language*, that serves as a module interconnection language and provides the glue for joining together computations spanning several megamodules. The language allows invoking megamodules, setting them up prior to invocation by supplying repetetive arguments to them, and extracting from them data and parameters, controlling their execution, transferring and transducing data between megamodules, and achieving asynchrony and parallelism of computations.

The underlying paradigm is that megamodules provide services. Megaprogramming has the potential of substantially increasing modeling power and software productivity by conceptualizing very large software systems in terms of interactions among very large components.

1. Introduction

The problem of scaling up software engineering to handle very large software systems is recognized as a bottleneck in software engineering [CSTB:90]. The term *megaprogramming* was introduced by DARPA (see: [BoSc:92]) to motivate research on this problem. This paper proposes a framework for megaprogramming in terms of software components called *megamodules* that capture the functionality of services provided by large organizational units like banks, airline reservation systems, and city transportation systems (see Figure 1).

Megamodules realize greater abstraction power than traditional modules by stronger encapsulation mechanisms: they can encapsulate not only procedures and data, but also types, concurrency, knowledge, and ontology. Programming within a megamodule can be handled by traditional technology, while computations spanning several megamodules are specified by *megaprograms* in a *megaprogramming language*. Megaprograms provide the glue for megamodule composition and typically involve communication over networks.

[1] Computer Science Department, Stanford University, Stanford, Ca 94305

[2] Computer Science Department, Brown University Providence, RI 02912

[3] Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milano, Italy

```
*******************************************************
* MEGAMODULE:                                         *
*   encapsulates data, behavior and knowledge         *
*   supports multiple concurrent activities           *
*   is autonomously operated and maintained           *
*   is a potential component of many megaprograms     *
*******************************************************
```

Figure 1: Properties of a Megamodule

In this paper we provide a framework for megaprogramming, recognizing that much work needs to be done to turn our vision into reality. The remainder of Section 1 introduces basic concepts and a running example. Section 2 outlines the general features and functionality of a megaprogramming language, which supports operations for invoking megamodules, for setting-up common arguments and extracting information, and for inspecting their state and monitoring their progress. Section 3 deals with the composition and interaction of megamodules, indicating how their compilation and optimization can take place. Section 4 describes an architecture for megaprogramming systems with libraries and databases, and deals with megamodule maintenance and life-cycle management. Section 5 positions megaprogramming relative to other work.

1.1 Encapsulation Power of Software Components

Software components provide their clients with services specified by their interface and encapsulate (hide) local structures that implement their services [Parnas:72], along the following progression:

- Functions and procedures: encapsulate statements and expressions.
- Objects and classes: encapsulate data and procedures.
- Megamodules: encapsulate behavior, knowledge, concurrency, ontology.

Functions and procedures encapsulate statements and expressions. Objects and classes realize greater expressive power by encapsulating data (instance variables) as well as procedures (methods) [Wegn:90]. Megamodules support effective hiding of domain-specific information, by encapsulating behavior, knowledge, and concurrency. A better encapsulation concept allows megamodules to model very large organizations as well as software communities having a local language, culture, and traditions.

Megaprogramming is a form of *programming in the large* that encompasses not only largeness of size but also persistence (largeness in time), diversity (large variability), and infra-structure (large capital investment):

- Size: many lines of code.
- Persistence: data survive the execution of programs.
- Diversity: of concepts, people, knowledge, traditions, software communities.
- Infrastructure: education, interfaces, tools, networks.

Traditional programming in the large, as defined in [DeKr:76], is concerned primarily with largeness of size, and does not substantially address the requirements of persistence, diversity, and infrastructure. To achieve the needed scale-up in software design and management, much current attention is placed on modeling the process and on enhancement of computer-aided software engineering (CASE) tools [Hump:88] [Blum:90].

Persistence requires great attention to the management of change. Diversity of concepts and people requires the coordination of diverse languages, multiple programming paradigms, alternate information representation, and heterogeneous components. An adequate infrastructure requires powerful tools and programming environments, as well as support of networks, system evolution, and education [BoSc:92].

The megaprogramming paradigm, modeling real-world services, is naturally parallel and distributed; it should support dynamic changes of the interface behavior and structure of constituent megamodules; for example, a bank may provide new services to its customers, while a corporation may create new manufacturing and service departments.

Megaprogramming is concerned with computations that span several software components. Its roots may perhaps be found in the work of [DeKr:76], which introduced *module interconnection languages* (MILs) in order to facilitate the construction and management of programs consisting of collections of software components, thereby recognizing that structuring collections of modules is an essentially different intellectual activity from constructing individual modules. Recent work ([BeLP:92]) on *module interface formalisms* (MIF) provides the basis for the specification and implementation of MILs.

Megamodules require stronger module interconnection mechanisms than traditional modules because of their stronger encapsulation facilities and their support of heterogeneous interfaces and dynamic evolution. Therefore, megaprogramming languages (MPL) extend MILs' expressive power, by handling information transfer between heterogeneous modules, dynamic queries and updates by users, distributed network communication protocols, and dynamically changing the specifications of interfaces. The traditional `CALL` statement becomes overloaded when faced with this array of tasks. Work on parallel languages has already replaced a single statement with multiple segments [Rinard:92]. We generalize on that concept. We present a particular MPL that is relatively low-level as a programming language but high-level when viewed as a networking language.

1.2 Ontologies

The term "ontology" derives from logic and artificial intelligence, where it denotes the primitive concepts, terminology, and interpretation paradigm of a domain of discourse [Gruber:91]. Whereas ontologies provide a conceptual framework for talking about an application domain and an implementation framework for problem solving. They provide a partitioning of context, so that systems that would otherwise be too complex to understand, become manageable [WRB+:90, Lenat:90].

Our paradigm for megaprogramming is based on partitioning of tasks into large components, called megamodules, that provide greater spatial, temporal, and conceptual independence than traditional modules. Each megamodule should have an internally consistent ontology, to reflect a uniform conceptual framework and problem-solving paradigm. Whereas ontologies in logic are specified by systems of axioms, the ontology of a megamodule should be determined by declarations specifying the properties of named entities accessible within the megamodule. Each declaration determines an *ontological commitment* to use the defined term in the manner specified by the declaration. The collection of all ontological commitments determines the ontology. Declarations may be classified by the nature of their ontological commitment into declarations for variables, operations, behavior, and knowledge:

```
1 declare(variable, x, 3)
2 declare(operation, successor, fun(x).(x+1))
3 declare(behavior, vehicleclass, vehiclebehavior)
4 declare(knowledge, time, ontology-of-time)
```

Variables, operations, and behavior can be introduced by traditional declarations. Ontological commitments for knowledge are beyond the current state of the art and beyond the scope of this paper, but are a central goal of artificial intelligence ([Gruber:91], [Sowa:91]).

Fortunately we can separate the question of ontological commitment within megamodules from that of communication among megamodules. Since we are largely concerned with megamodule composition rather than with their internal structure, the mechanisms of ontological commitment, though important to the practical realization of megaprogramming, are not a primary concern of this paper: the specification of ontologies local to a component remains a local issue.

1.3 An Example: Transportation of Goods Between Two Cities

As an illustration, consider the transportation of goods between two cities. The general problem is that of shipping goods between any pair of cities by various forms of transportation, such as air, rail, or truck. Local transportation within each city is managed by megamodules, and intercity transportation by air, rail, and truck is managed by other megamodules. The complexity of each megamodule is substantial and the total number of megamodules can be moderately large. We examine a specific application in greater detail.

Consider a logistics application for the shipment of goods between two U.S. Navy installations, NOSC in San Diego to NRL in Washington. This Ship-Goods task can be realized by a megaprogram using three megamodules for:

1 Surface transport in San Diego from NOSC.
2 Commercial air freight.
3 Surface transport in Washington DC from one of its airports to NRL.

The megamodules (1) and (3) will be similar in structure and processing, but will use different databases. Local experts in these cities will maintain their own databases. Megamodule (2) will differ in ontology as well, as it represents and processes data according to its particular needs and according to the unique characteristics of the environment.

Megamodule (1), which deals with San Diego surface transport, will use tables that enumerate the trucks available to NOSC, the times when they are available, their capacities, and their loading facilities. It will also contain a San Diego roadmap and transit times for road-segments at different times of day. Programs within this megamodule will deal with reserving trucks, moving them to NOSC, their loading, and travel to the airport. They will also be able to produce cost and time estimates in these tasks. However, when the departure time from NOSC is not known, the estimates may have excessive uncertainty or require very large tables of alternatives for various conditions.

Megamodule (2), for planning delivery of goods via commercial air freight, is itself a substantial application. It has to be able to choose among multiple airlines, obtain schedules and tariffs, and check for available space. This megamodule may itself invoke computational codes shared by other megamodules through conventional interfaces to program libraries.

Megamodule (3), for the Washington surface-transport, will need to know routes from each of the three Washington airports. Rush hours differ among cities, and in Washington some critical roads are closed in one direction during rush hours. Geography makes a difference as well. Heuristics that are effective in San Diego, such as 'take a local road if the main road is blocked', fail when dealing with Washington bridges.

The megaprogram will then coordinate the activities of megamodules (1), (2), and (3) and present to its end-user the *best* plan for the shipment. The execution sequence of megamodules may be affected by a variety of application requirements and constraints. If the *time-of-arrival* at NRL is the critical factor, then execution should probably commence with the Washington megamodule and proceed backwards. If instead the *time-ready* of the shipment at NOSC is supplied to the San Diego megamodule, then forward computation determines the arrival time. Without such requirements, the megaprogram might focus first on the most economical flight.

```
                  End User A                End User B
                 --------------            --------------
                 | IO module  |            | IO module  |
                 | suitable   |            | suitable   |
                 | for user A |            | for user B |
                 --------------            -------------
                       ||                        ||
              -----------------------------------------------
              |         Ship-Goods Megaprogram              |
              -----------------------------------------------
                       ||              ||              ||
                       || (1)          || (2)          || (3)
                       ||              ||              ||
 -------------- -------------- -------------- -------------- --------------
 | Megamodule | | Megamodule | | Megamodule | | Megamodule | | Megamodule |
 |  warehouse | | surface tr.| | airfreight | | surface tr.| | ocean tran.|
 | (not used) | | San Diego  | |  Com US    | | Washington | | (not used) |
 -------------- -------------- -------------- -------------- --------------
      [ lower-level megamodules and databases ]
```

Figure 2. Example of use of megamodules by a megaprogram.

2. Megaprograms and Megaprogramming Languages

Megaprograms compose and schedule the computations performed by megamodules. Mega-programming languages must allow flexible composition of megamodules and support both synchronous and asynchronous coordination schemes, decentralized data transfer, parallelism, and conditional executions.

In this section, we propose a particular MPL [WWC:90]; many concepts presented here, though specifically related to one proposed language, are quite general. The language is used for introducing functionalities to be implemented by megaprograms and megamodules. The reader should be aware that the language and its functionalities correspond to our vision; research and development is required to turn this vision into reality. The design principles of MPL can be summarized as follows:

- Traditional `CALL` statements for invoking remote procedures and passing parameters, are overloaded when used to compose large systems. MPL segments their functions by separating parameter management in input and output from the invocation.
- Flexibility and asynchrony is enhanced, by allowing autonomous execution of megamodules, while the invoking megaprograms retain great flexibility, as MPL statements can monitor, inspect, and constrain the execution of megamodules and can accept partial results.
- We are inspired by our database background; databases provide services autonomously; input and output parameters of megamodules are presented through database-like schemas, and the ideas behind megamodule optimization have several common features with query optimization.

The distinguishing feature of the proposed MPL is the substitution of the traditional `CALL` statement, which exerts control and provides communication between software components, by three distinct statements: `SET-UP`, `INVOKE`, and `EXTRACT`.

a. `SET-UP` provides global arguments from a megaprogram to the megamodule.
b. `INVOKE` causes the megamodule to actually process these arguments and prepare results.
c. `EXTRACT` permits the megaprogram to actually extract results from the megamodule. Further statements are used for inspecting and controlling megamodule execution. The most important are:
d. `EXAMINE` to allow the megaprogram to check on the progress being made in obtaining results.
e. `ESTIMATE`, to cause the megamodule to provide a performance estimate, so that the megaprogram can judge whether invocation is likely to be effective.

To illustrate our MPL, we give a simple megaprogram that fills an order by invoking two megamodules "`Producer`" and "`Consumer`", supplying and extracting data for each megamodule, and transducing the output of the producer into the format required by the consumer. Keywords of MPL are entirely in capital letters, megamodule and megaprogram names have an initial capital letter, and datanames are entirely in small letters.

```
MEGAPROGRAM Fill-order(Producer, Consumer)
  Input product, list-of-parts
  SET-UP Producer, list-of-parts
  INVOKE Producer.product
  EXAMINE (Status) UNTIL Status = DONE
  EXTRACT Producer, list-of-produced-items
  PERFORM Transduce(list-of-produced-items, list-of-consumer-items)
  SET-UP Consumer, list-of-consumer-items
  INVOKE Consumer.purchase
  EXAMINE (Status) UNTIL Status = DONE
  EXTRACT Consumer, list-of-purchased-items
  Output cost, list-of-purchased-items
```

Figure 3. A Simple Megaprogram

Our MPL is not complex from a programming language point of view, but is higher-level than a module interconnection language. In the following, we examine the features of our MPL in greater detail; we present first the data structures for the set-up and extract operation, next the operations for megamodule interaction; then the operations for inspecting at compile-time the interfaces and content of megaprograms; and finally, those for examining the run-time status of computations. Compile-time and execution-time can easily overlap.

2.1 Data Structure Interfaces for Set-Up and Extract

The *data structures* supported by a megaprogramming declaration enable providing input and output parameters, of arbitrary complexity, between a megaprogram and megamodules; data structures are defined in the context of megamodules and are accessible by megaprograms. The MPL type system should be powerful enough to support complex data structures; therefore, it should include generalized type constructors, including records, sets, multisets, lists, sequences, updatable arrays, sparse arrays, etc.

Data structures of megamodules are designed independently; they have arbitrary internal type systems and are defined asynchronously from megaprogram definitions. Megamodules also provide self-describing `EXPORT` data structures that should be compatible with the MPL type system. However, a megamodule need not to be able to handle all of the MPL type system, its capability must only be adequate to support all of its `EXPORT` data structures.

The technology we propose to borrow and adapt here derives from database schemas, the description of database contents and formats that permits many transaction programs to share a database and obtain data selectively and associatively. Accordingly, datastructures of megamodules are declared in terms of a schema definition, but significant extension of database schema concepts will be required, since:

1 The data structures permissible in general megaprograms are more complex than the simple relations seen in most DBMSs [CCZLL:90].

2 Enough information must be provided to permit transduction of information between data structures that differ. Transduction occurs when data extracted from a megamodule are submitted to a successor megamodule: optimizations, described in Section 4, combine consecutive `EXTRACT` and `SET-UP` operations to generate a direct flow of information among megamodules.

2.2 Operations for Megamodule Interaction

These operations split the traditional "`CALL module;`" statement into three parts, to allow adequate management of large megamodules. This split also fosters asynchronous operation and parallel execution of megamodules, as depicted in Figure 4.

SET-UP The `SET-UP` statement provides information to the megamodule as a subset of the megamodule's data structure. When a set-up statement is compiled by the megamodule, the type of the MPL data structure is compared with the type of the datastructure definition previously exported by the megamodule; the megamodule is responsible for managing type coercion transformations needed for ensuring compatibility of internal data structures with the MPL type system. Problems of this translation inside a megamodule are not discussed in this paper.

The set-up of data values by megaprograms conforming to this general schema may differ. In particular:

- Many argument values may not be supplied explicitly by the megaprogram; for these default values are assumed by the megamodules. These defaults are set when `SETY-UP` has not supplied values.
- Some elements of the `SET-UP` data structures may be defined to be `PARAMETER`s. Such parameter values are specified with the actual invocation of the megamodule. During execution of the `SET-UP` statement, the megamodule retains control. This assures that the data transmission to the megamodule is synchronized properly.

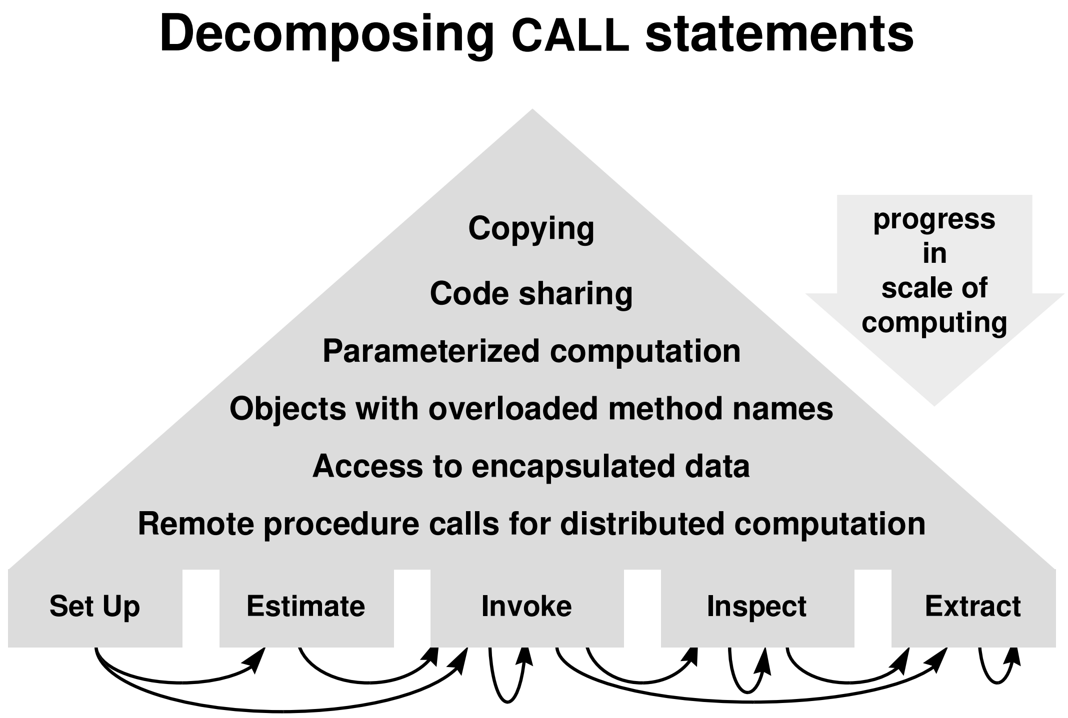

Figure 4. Modules in a mediated architecture and the scope of an application.

INVOKE The statement "`INVOKE megamodule WITH parameters`" causes the initiation of asynchronous execution of the megamodule with any parameter values listed in the `WITH` portion. The parameter values are bound to those datastructure items that were defined as parameters in the `SET-UP` statement. After the `INVOKE` statement is processed, control is returned to the megaprogram; thus, the megaprogram can schedule other work in parallel.

EXTRACT An `EXTRACT` statement provides megamodule results to the megaprogram. All variables that are expected to be extracted as result of the megamodule execution are made available to the MPL environment as datastructures accessed through this MPL statement. The result is fully valid when the megamodule state is `DONE` (corresponding to completion of the megamodule activity, as shown in Figure 3), but we place no constraints on earlier extraction of incomplete information.

During execution of the `EXTRACT` statement, the megamodule again retains control. This assures that in this phase, as during the execution of `SET-UP`, the data transmission is synchronized by the megamodule.

Note that `INVOKE` is typically executed asynchronously, with the parameters of the invoke being part of the activation message; megamodule operation can, however, remain independent. For instance, a megamodule may always be active and keep the `EXTRACT` data structure up-to-date. Then, the `INVOKE` message only provides a synchronization handle and the `EXTRACT` operation may be immediately executed. A megamodule might be active periodically based on external triggers. An `EXTRACT` operation then provides potentially not up-to-date snapshots; the `EXTRACT` operation may be executed at any time, since it does not affect the behavior of the megamodule.

2.3 Operations for Inspecting the Interface and Content of Megaprograms

The following operations, provided by megamodules, are *public* for use in a megaprogramming environment; they can be requested by megaprogrammers and their result helps in the specification and design of megaprograms. They can also be used by MPL compilers to obtain information and may be used at execution-time in interpretive situations. These operations permit megamodules to specify the data structures and defaults that they import and export; some megamodule-specific information can also be made available for inspection. Note that the information flows up, while traditionally subroutines depend for their environment on the invoking programs, and thus are difficult to reuse.

IMPORT SET-UP The `IMPORT SET-UP` statement causes the megamodule to export the set-up data structures. Standard names are used to describe data structures, formats, extents, and sizes, according to MPL types. Elements of data structures may have default values which can be retrieved by an `INSPECT` command.

IMPORT EXTRACT The `IMPORT EXTRACT` statement, similar to the `IMPORT SET-UP`, causes the megamodule to export the extract data structure.

INSPECT Inspection of a megamodule provides information about its ontology and specific parameters; the result of an `INSPECT` operation does not depend on the current state of the module being inspected. Inspection could yield documentation, version information, or formal descriptions of methods and scope of applicability.

2.4 Operations for Examining the Status of Megamodules

A number of operations permit interaction of megaprograms and megamodules. Their function is to aid in execution to attain a high level of efficiency. They may, for instance, provide information that leads to a rescheduling of operational sequences of megamodule invocation. Since we assume a parallel computing environment throughout, such flexibility is essential to exploit parallel capabilities in a dynamic manner.

EXAMINE In addition to the data structures used to set-up and extract data, a megamodule maintains a number of state variables. The `EXAMINE` statement accesses the most critical variable, namely `STATE`. Reasonable values for `STATE` variables in megamodules are: `DONE`, `IN_PROGRESS`, `IDLE`, `IN_ERROR`, or `WAITING`. For instance, `STATE=DONE` indicates that the megaprogram can safely extract results from the megamodule. Other variables that may be provided by a megamodule include, for instance, measures of nearness to a solution for an iterative program, the number of choices being considered for a search-type megamodule, and so on. These variables may be accessed at any time for a megamodule that has been invoked. The operation of `EXAMINE` is synchronous with respect to the megaprogram.

ESTIMATE For megaprogramming optimization, estimates of execution cost of megamodules may be needed [Wie:92a,Wie:92b]; because of the encapsulation provided by megamodules, such estimation can only be provided from inside the megamodule. The result of execution of an `ESTIMATE` statement is a simple estimate of execution cost (time, dollars, result volume), with the degree of certainty of that estimate (0..1).

CONSTRAIN The `CONSTRAIN` statement permits the resetting, by a megaprogram, of internal megamodule parameters obtained by inspection.

LIMIT The `LIMIT` function constrains megamodule execution in terms of time or cost (as a surrogate for computational time spent). Not all megamodules will be able to accommodate such a request, but if they can, certain megaprogram tasks would be greatly simplified.

```
-------------------------------------------------------------
|                      MEGAPROGRAM                          |
|          compilation                         execution     |
-------------------------------------------------------------
          **        **      *        | |        |  *  ***
          || IMPORT ||      |        | | SET-UP |  |  | | EXTRACT
          || META-  ||      |        | | DATA   |  |  | | DATA
          ||  DATA  ||      |        | |        |  |  | |
          ||  for   ||      |        | |        |  |EXAMINE
          ||SET-UP  ||      |INSPECT | |  INVOKE|  |  | |
          || EXTRACT||      |       ***         *  |  | |
        ---------------------------------------------------
        |                  MEGAMODULE                     |
        |            Exported datastructures              |
        |            Mappings to local values             |
        |            Local datastructures                 |
        |            Local programs and databases         |
        ---------------------------------------------------
```

Figure 5. Information flow in megaprogamming.

3. Composition and Interaction of Megamodules

In this section we discuss the composition and interaction of megamodules as it is achieved through compilation and optimization of megaprograms, as sketched in Figure 5.

3.1 Composition and Interaction of Traditional Software Components

Function composition in traditional languages can be specified by the notation "$f(g(x))$", where f operates on the value produced by "$g(x)$". Procedure composition is realized by executing a sequence of procedure statements "$P; Q; R$" where each transforms a global state.

The composition of megamodules requires more than simple catenation. Information has to be transduced: collected, transformed, and forwarded among megamodules and input-output modules. To elucidate these MPL concepts we can compare how direct communication among programs is specified in a UNIX environment. MPL concepts can be viewed as a generalization of such UNIX concepts.

The UNIX shell plays the role of a megaprogramming language for UNIX processes. UNIX pipes provide direct communication among processes that transform a stream of input data into output data. A UNIX command of the form "$P1 \mid P2$" allows the results of the process $P1$ to be incrementally piped to $P2$, so that $P2$ can process the output stream of $P1$ in parallel with further computation by $P1$.

The generalizations provided by MPL require that we identify each MPL operation with an explicit megamodule name, since the compositions of `IMPORT`, `SET-UP`, `INVOKE`, `EXAMINE`, `EXTRACT`, etc., are arbitrary and any of the megamodules may be addressed. In MPL, megaprogram execution will be asynchronous, similar to what UNIX supports with the form "$P\&$". However, the megamodules remain accessible through the `EXAMINE` statement, so that their progress can be monitored. The combination of asynchrony and control adds much power to the megaprograming language paradigm. Research will be needed to phrase a syntactically attractive language which permits specification of the parallelism and asynchrony implicit in the megaprogramming paradigm.

3.2 Composition and Interaction of Megamodules

A simple pattern of interaction between the megaprogram and megamodules can be realized by `SET-UP`, `IN-VOKE`, and `EXTRACT`. Note that even this simple pattern of interaction improves the conventional development of applications. The `SET-UP` and `EXTRACT` data structures provide a higher level of abstraction with respect to direct linking of software modules, enabling transduction of data structures between megamodules according to mechanisms which are defined in the megaprogram. Selective specification of parameters reduces set-up and communication costs.

More complex interaction, yet still under the direct control of the megaprogrammer, is possible through the use of control structures `IF ... THEN ...`, whose effect is based on returned parameters from the `INSPECT`, `IMPORT`, `EXAMINE`, and `ESTIMATE` statements. By being able to assess the behavior of a megamodule in execution, it is possible to achieve important flexibility at execution time. In particular, execution of an unproductive module may be aborted and a different execution pattern may then be tried. When computational resources are plentiful, multiple similar megamodules may be started in parallel, and only the one giving the best response be retained. This approach requires that the `INVOKE` primitive be nonblocking, and also requires explicit time control within the MPL. The MPL will need an internal time-driven interrupt, so that it can decide when situations equivalent to `TIME-OUT` in conventional control of independent processes occur.

The access to a megamodule might be subject to limits, and the *megaprogram composer* might be able to check that limits are feasible through the `ESTIMATE` command. This merges the traditional compilation and execution phases into a complex interaction.

3.3 Compiling Megaprograms

Compilation of megaprograms is important to reduce the cost of extract, transduce, and set-up sequences among modules. This cost is especially high if the megamodules are widely distributed. For compilation and initial optimization all interactions require communication with the site where the megaprogram resides. After compilation, information can be directly conveyed among the megamodules, and the load on the network as well as on the originating site of the megaprogram can be considerably reduced.

Many of the meta-information statements, previously described as aiding the execution of an interpreted megaprogram, will be used by the megaprogram compiler to provide the input for compilization decisions. Compilation enables *megaprogram optimization*, where the entire megaprogram is submitted to an optimization module. Such optimization would be responsible for:

- Combining consecutive `SET-UP` and `EXTRACT` operations by generating a direct flow of information between megamodules.
- Deciding the order of invocation of megamodules, in particular invoking them in the "optimal" (e.g., fastest, least expensive, ...) order.
- Introducing parallelism and asynchrony in the schedule of invocations of modules.

These optimizations might seem novel in the field of software engineering but have been successful in database systems. The information provided by `INSPECT`, `EXAMINE`, `ESTIMATE` is comparable to the information available to a database query optimizer.

The composition for our initial example, in the case that the `ready-time at` NOSC is given, might consist of:

1 Invoking the San Diego module with information about the `ready-time` and the load. Extracted from that module would be cost and time-required information, and any needed schedule and routing data. Now the earliest arrival-time of the load at Lindberg field in San Diego can be computed.

2 The air-freight megamodule is now supplied with these specifications, invoked, and similar results are extracted. The arrival-time at, say at Washington Dulles airport, for the best flight, is now computed.

3 The Washington surface module is now invoked with that time and the final arrival-time at NRL is now produced.

4 Desired output, extracted from these modules, is supplied to an output module by the megaprogram.

Many versions of such megaprograms can be envisaged. A megaprogram may iterate through the alternate airports, given as choices by the air-freight module, to optimize the Washington end of the transport task. Also, as indicated earlier, if the due-time at NRL is the critical factor, then the megamodules will be invoked in the reverse order.

```
 SET-UP Surface-transport       --  set up megamodule
       ( Goods (Weight = 5000 UNIT kg,
               Volume = 30 UNIT m3,
               Maxunitsize = 5 UNIT m,
               ....)
         From (Town = San Diego, Loc = PARAMETER locn)
         To   (Town = Washington, Loc = NRL, building22)
         At   (Date = 12JUN90, Time = PARAMETER T_ready )
               ....)
INVOKE Surface-transport       --  start megamodule
         WITH (T-ready = 18:00,
              locn = (NOSC, building 17),
              ....)
```

Figure 6. Example of a megaprogram invoking one megamodule.

If minimal cost is the criterion, step 2 will likely gain priority and be scheduled first with minimal constraints.

4. Megaprogramming System Architecture

The architecture of a megaprogramming system consists of a collection of geographically distributed megamodules linked by high capacity networks; the megaprogramming environment should include a repository and dictionary, and support megamodule execution and maintenance.

4.1 Megamodule Repository and Dictionary

Information about the collection of megamodules in the megaprogramming environment is specified in a data dictionary that includes the following information about each megamodule:

1. The names of megamodules which are currently available.
2. A description of the interface of the megamodule (simply stated, some formal description of the function performed by the module).
3. Information about data to be supplied in the set-up and available for extraction, expressed as data structures in the MPL type system.
4. Internal information about megamodules; these include the programming language used by the megamodule, its internal modularization, the schemas of databases which are accessed by the megamodule, the level of consistency provided by the megamodule applications in accessing the databases, real-time constraints on megamodule execution, etc.
5. An indication of the availability and the cost of using the megamodule.

Database techniques may provide useful concepts in organizing this information and making it available to the megaprogrammer through conventional and ad-hoc query interfaces. In particular, the `IMPORT SET-UP`, `IMPORT EXTRACT`, and `INSPECT` statements in MPL may be considered as formal interfaces to this dictionary.

4.2 Support for Megaprogram Execution

The data dictionary (library) is part of a program execution environment which may be replicated at many sites, including sites associated with individual megamodules, to permit distributed specification and execution of megamodules. The responsibilities of the megaprogramming environment in supporting execution include:

1. Type checking the consistency of `IMPORT` pairs for successive `EXTRACT` and `SET-UP` operations. If the type systems are different, provide required routines for type coercion.
2. Generate executable code for megaprograms after compilation, where this code might consist of several remote procedure calls and support for data transduction.
3. Generate compiled versions of the megamodules which are used in the context of a specific megaprogram, with optimizations due to the values provided in the `IMPORT` data structure.
4. Provide late type checking for run-time parameters.
5. Handle concurrent execution and synchronization of multiple megaprograms, initiated at different sites of the megaprogramming system.
6. Deal with the notion of transaction atomicity in the context of megaprograms.

4.3 Access to Databases

We assume that databases are part of megamodules. The semantics of the database being used is completely captured within the "ontology" of the megamodule and does not need to become "public." Instead, datastructures for `SET-UP` and `EXTRACT` do become public; they will be influenced by the schemas of databases which are stored within a megamodule.

We do not focus on fully transparent distributed databases where access to any data item can be made from any site regardless of the logical fragmentation and physical distribution; this issue is not the concern of megaprogramming. If used, their locking schemes will constrain megamodule execution parallelism. Our view is consistent with autonomous and independent access; we can lower consistency requirements to match those see in federated databases [Litw:86], because we do not expect to provide comparable semantics across databases which are managed within different megamodules.

4.4 Transaction Management and Concurrency Control

We safely assume that durability (e.g., permanence of effects of committed transactions), local serializability (i.e., equivalence of each local schedule to some serial local schedule), and isolation will be provided by each local database management system. Therefore, the only issues concern global atomicity and serializability of megaprograms, seen as large transactions operating as a unit over disperse, heterogeneous databases. This problem is quite hard, but some practical solutions have been studied; they are briefly mentioned in Section 5.3.

4.5 Maintenance

A major motivation for our approach to megaprogramming is to improve the management of maintenance tasks. The development and maintenance of each megamodule is entirely under the authority of a local software development team. This provides freedom for independent development and maintenance of each megamodule, but also confers a responsibility for providing reliable service to clients. Reliability can be ensured by globally enforced *acceptance testing* for both new modules and new versions (releases) of already existing modules. Some megamodules may have requirements for continuous execution of processes that monitor ongoing real-world activities, and cannot be taken out of service.

Maintenance is greatly facilitated by the separation between megamodules and megaprograms. Maintenance is partitioned at both the megaprogram and megamodule levels.

- Maintenance within each megamodule is done by an expert of the megamodule's "ontology." We distinguish two types of changes. *Local changes* do not affect the `SET-UP/EXTRACT` data structures, and therefore do not require megaprogram recompilation. *Global changes* affect the `SET-UP/EXTRACT` data structures, and therefore cause megaprogram recompilation.
- Maintenance of a megaprogram is done by megaprogrammers; this may be due to changes of the invocation strategies of megamodules, or might be due to global changes to megamodules.
- No maintenance operation should span two megamodules.
- While the maintenance of a specific megamodule is taking place, it is advisable to support the old version together with the new version, so that use of existing megaprograms can continue.

Observe that the rules for maintenance differ when megaprogramming:

Traditional: A maintenance programmer can touch any module of a project

Megaprogramming: A megamodule programmer can only touch a single megamodule.

5. Comparison With Related Work

Megaprogramming is primarily a software engineering activity, but borrows heavily from the technologies of databases and of artificial intelligence.

5.1 Software Engineering

The goals of megaprogramming, developing a component-based software technology for programming in the large, are also central goals of software engineering. Megaprogramming builds on earlier work on data abstraction [LisGu:86, Parnas:72], object-oriented programming [GolRob:83], and distributed computing [CoDo:88].

Scaling up from objects to megamodules goes beyond object-oriented notions of encapsulation and message passing; techniques for managing megamodules and communicating between them are qualitatively different from those for objects. Because megamodules are so much more substantial than traditional modules, techniques for object management through types, classes and inheritance are less applicable; megamodules require special database retrieval, browsing, and instrumentation methods.

Reusability is another related area in which traditional mechanisms [BiPe:89] need to be supplemented to handle the larger granularity of megamodules. We have defined megamodules as the unit of use and reuse by megaprograms in a manner that use and reuse become indistinguishable. Removing the modules from the invoking environment is critical.

Kron and DeRemer's [DeKr:76] and subsequent work [BeLP:92] on module interconnection languages and programming in the large is very influential, though megamodule interconnection mechanisms differ from traditional point-to-point communication mechanisms of object-based languages.

The need for a distinct language to manage module distribution alternatives of an application was recognized by Purtilo in his thesis and implemented in his POLYLITH software bus [Purt:90]. Our megaprogramming paradigm extends this view by proposing a compilable language for information flow control, conversion, and optimization (*transduction*), avoiding programmer involvement as system configuration and application needs change.

5.2 Databases and Transaction Processing

Work in databases has taken a number of recent directions [Silber:91] that are relevant to megaprogramming, including enhanced functionalities for data distribution and richer data models through the use of rich type systems for data definition. In this paper, we suggest that the ideal data model for megaprogramming should have a rich type system, but we deliberately don't take a position on whether it must be object-oriented; indeed, the usefulness of concepts such as object identity and generalization hierarchies with distributed data, autonomously managed by each megamodule, is currently being debated.

Databases must already deal with heterogeneity, sometimes caused by finding that distinct information resources are located at distinct sites and under distinct management. The query program has then no control over the internals of the resource representation, and very little control over the interface. These systems go by the name of *federated systems*. Languages to deal with combining information from federated systems have been developed [Daya:85] [Litw:86].

Transaction processing is another field of concern for megaprogramming. Classical transactions are minuscule tasks which only interact with each other through the database; typical workloads reach thousands of these small transactions per second. However it has become clear that not all database applications fit into this basic paradigm, so that new models of transactions are emerging, including *long transactions* [Katz:83] and *nested transactions* [Moss:85]. Long-lived transactions typically release their locks and run at a lower level of consistency with respect to full serializability. *Sagas* [Garcia:87] are particular nested transactions which achieve global atomicity but permit the autonomous commitment of component subtransactions, whose effect can be undone by compensation (typically, by firing another transaction that will reconstruct the initial state). These mechanisms may find an application in megaprogramming, where the autonomy of each megamodule is a strict requirement.

5.3 Artificial Intelligence

Independence of modules has been a long-standing concern in artificial intelligence (AI). Specifically the paradigm of independent actors and agents has captured attention from [Hewitt:73] to [Genesereth:89], proposing various models for building complex computations and knowledge bases starting from knowledge which is independently provided in different contexts. Since programming was not an objective of AI researchers, this work was focused on bringing intelligence within modules and on providing loose and flexible control of module interconnection. In contrast, we propose mechanisms for megamodule control that, though maximally flexible, are fully specified through the MPL instructions; we do not expect that megamodules propose and negotiate their own computations, as is often expected in AI models.

A broader view is taken by people working in Enterprise Integration The CARNOT project at MCC is using a single knowledge-base, within CYC ([Lenat:90]) to manage diverse resources [CWST:92]. The language they have adopted, however, was used to describe homogeneous parallel operations and does not deal directly with the diversity found.

The notion of megamodule ontology and the importance of ontological commitment (i.e., that entities within megamodules conform to their declarations) was developed in the context of knowledge representation models [Gruber:91]; megaprogramming borrows this notion in order to define the expected qualities of megamodules, which however remain out of megaprogramming control.

6. Conclusion

We have defined megaprogramming as a departure from standard top-down programming practice, which has not been able to deal well with the problems encountered when building and maintaining large software systems. Our objective has been to provide a conceptual framework for the study of megaprogramming and to identify technical problems to be addressed in developing an effective technology of megaprogramming.

Concrete application of a comprehensive megaprogramming technology along the directions outlined in this paper is clearly still far away. A number of problems need to be solved, not only related to megamodule integration, but also in the building of individual components, especially if we aim at enhancing their portability and reusability. Improving "programming in the medium" is a premise for the effective development of megaprogramming. Some of its principles can be incorporated in handcrafted systems.

Another critical factor to the success of megaprogramming is its integration with current standardization efforts. Emerging standards for supporting remote database access and distributed commitment protocols may provide useful concepts for the technological infrastructure needed for supporting megaprogramming requirements on a computer network (based on transfer and transduction of data structures among megamodules).

Though we see an existing trend toward the design of powerful MPLs and module interface formalisms, we have to significantly raise the level of expected functionalities compared to the current state-of-art; feasibility and applicability of these ideas must be demonstrated through further research and experiments. This paper identifies a number of open research issues:

- Design of a megaprogramming language and type system.
- Design and implement the megaprogramming support environment.
- Design optimizers for megaprograms, with an increasing degree of sophistication, including the minimization of coercion and transduction costs.
- Establish trade-offs between pre-execution compilation and dynamic scheduling of megaprogams.
- Understand the role of emerging standards in the context of databases and transaction processing systems for the implementation of megaprogramming interfaces.
- Understand deeply the nature of "ontologies" as a basis for programming megamodules.
- Understand and disclose the methodological issues concerned with building individual megamodules so as to assure their reusability in the development of new applications.
- Understand and disclose the methodological issues concerned with building complex megaprogramming applications. The switch to a service from a subroutine paradigm refocuses many of these issues.

Acknowledgement

We have received useful comments on this work from Tom Dean, Tom Doeppner, Steve Reiss, and Peter Rathmann. We are grateful to the anonymous referees for their constructive comments and suggestions. A long-standing interest in new programming paradigms has been encouraged by Sheldon Finkelstein, Witold Litwin, Carl Hewitt, Yoav Shoam, and others. Work of Stefano Crespi-Reghizzi, Letizia Tanca, and Roberto Zicari contributes insight into the applicability of these concepts. The initial motivation for this formulation was provided at the 1990 DARPA-ISTO Software Enginering PI meeting, where the need for new directions in programming-in-the-large was explicated and the term *megaprogramming* appeared.

Background research for work has been partially supported by the DARPA contract N39-84-C-211. Earlier, Gio Wiederhold was supported by IBM Germany. Recent support includes a grant on coordinating updates to replicated copies on federated databases (FAUVE Contract IRI-9007753, NSF), a grant on Knowledge-based Mediators from the IBM Knowledge Systems Lab, Menlo Park, and a DARPA contract to ISI for development of standards for Knowledge-querying and -manipulation. Stefano Ceri is partially supported by the ESPRIT Projects "Stretch" and "Idea" and by the CNR Project "Logidata+."

References

[BeLP:92] F. C. Belz, D. C. Luckham, and J. M. Purtilo: "Application of ProtoTech Technology to the DSSA Program"; *Proc. DARPA Software Technology Conference 1992*, Los Angeles CA, April 28–30, Meridien Corp., Arlington VA 1992.

[BiPe:89] T. J. Biggerstaff and A. J. Perlis: *"Software Reusability"*; Addison-Wesley (ACM Press) 1989.

[Blum:90] B. I. Blum: "Modeling-in-the-Large"; draft of Chap. 3 of 'Software Engineering for Large and Small Projects', Milton S. Eisenhower Research Center, Johns Hopkins University, T.R.RMI-90-023, Sept.1990.

[BoSc:92] B. Boehm and B. Scherlis: "Megaprogramming"; *Proc. DARPA Software Technology Conference 1992*, Los Angeles CA, April 28–30, Meridien Corp., Arlington VA 1992.

[CCZLL:90] S. Ceri, S. Crespi., R. Zicari, G. Lamperti,G., and L. Lavazza: "ALGRES: an advanced database system for complex applications"; *IEEE Software*, Jul.1990.

[CoDo:88] G. F. Colouris and J. Dollimore: *"Distributed Systems: Concepts and Design"*; Addison Wesley 1988.

[CSTB:90] Scaling Up: A Research Agenda for Software Engineering, Summary of a Report by the Computer Science and Technology Board (CSTB); *Communications of the ACM*, 33:3, March 1990.

[Daya:85] U. Dayal: "Query Processing in a Multidatabase System"; in *Query Processing in Database Systems*, 1985, Springer, pp.81–108.

[DeKr:76] F. DeRemer and H. H. Kron: "Programming in the Small Versus Programming in the Large"; *IEEE Transactions on Software Engineering*, June 1976. Also in *Tutorial on Software Design Techniques*, Eds P. Freeman and A. I. Wasserman, IEEE Computer Society Press 1983.

[Garcia:87] H. Garcia-Molina and K. Salem: "Sagas"; *Proc. ACM-SIGMOD Int. Conf. On Management of Data*, S. Francisco, 1987.

[Genesereth:89] M. R. Genesereth: "Proposal for Research on Informable Agents"; Logic Group Report 89-9, Stanford University, May 1989.

[GolRob:83] A. Goldberg and D. Robson: *"Smalltalk, The Language and its Implementation"*; Addison-Wesley, 1983.

[Gruber:91] T. R. Gruber: "The Role if Common Ontology in Achieving Sharable, Reusable Knowledge Bases"; in Allen, J.A., Fikes, R., and Sandewall, E. (Eds) *Principles o f Knowledge Representation and Reasoning*, Morgan-Kaufmann, 1991.

[Hewitt:73] C. Hewitt, P. Bishop, and R. Steiger: "A Universal Modular ACTOR Formalism for Artificial Intelligence"; *IJCAI* 3, SRI, Aug.1973, pp.235-245.

[Humphrey:88] W. S. Humphrey: "Characterizing the Software Process"; *IEEE Software*, Vol.5 No.2, March 1988, pp. 73–79.

[Katz:83] R. H. Katz: "Managing the Chip Design Database"; *IEEE Computer*, Dec.1983, pp.16–36.

[Lenat:90] R.V. Douglas Lenat et al.: "CYC, Towards Programs with Common Sense"; *CACM*, Vol.33 No 8, Aug.1990.

[LisGu:86] B. Liskov and J. Guttag: *"Abstraction and Specification in Programming Languages"*, MIT Press 1986.

[Litw:86] W. Litwin and A. Abdellatif: "Multidatabase Interoperability"; *IEEE Computer*, Vol.19 No.12, Dec.1986, pp.10–18.

[Moss:85] J.Eliot B. Moss: *Nested Transactions, an Approach to Reliable Distributed Computing*; The MIT Press, 1985.

[Parnas:72] D. L. Parnas: "On the Criteria to be used in Decomposing Systems into Modules"; *Communications of the ACM, 1972.* Also in *Tutorial on Software Design Techniques*, Eds P. Freeman and A. I. Wasserman, IEEE Computer Society Press 1983.

[Purt:90] J. M. Purtillo: "The Polylith Software Bus"; Univ. of Maryland Tech. Report 2469, 1990; to appear in *ACM Transactions on Programming Languages and Systems.*

[Rinard;92] M. C. Rinard and M. S. Lam: "Semantic Foundations of Jade"; *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages* Jan. 1992.

[Silber91] A. Silberschatz, M. Stonebraker and J. Ullman (eds.): "Database Systems: Achievements and Opportunities"; *Communications of the ACM*, 34:10, Oct. 1991, 110-120.

[Sowa:91] J. F. Sowa: "Issues in Knowledge Representation" in J.F. Sowa (ed.) *Semantic Networks: Explorations in the Representation of Knowledge*, Morgan-Kaufmann, 1991.

[Wegn:90] P. Wegner: "Concepts and Paradigms of Object-Oriented Programming"; *Object-Oriented Messenger*, Vol 1, Number 1, August 1990.

[WRB+:90] G. Wiederhold, P. Rathmann, T. Barsalou, B. S. Lee, and D. Quass: "Partitioning and Composing Knowledge"; *Information Systems*, Vol.15 no.1, 1990, pp.61-72.

[Wie:92a] G. Wiederhold: "Mediators in the Architecture of Future Information Systems"; *IEEE Computer*, March 1992, pages 38–49.

[Wie:92b] G. Wiederhold: "Model-free Optimization"; *Proc. DARPA Software Technology Conference 1992*, Los Angeles CA, April 28–30, Meridien Corp., Arlington VA 1992, pp. 82–96.

[WWC:90] Gio Wiederhold, Peter Wegner, and Stefano Ceri: "Towards Megaprogramming"; Stanford University, Report No.STAN-CS-90-1341 Brown University, Report No.90-20, Oct.1990, and Politecnico di Milano, Dipartimento di Elettronica, N. 90-055.