Towards Megaprogramming

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

Technical Report No. CS-90-20
October 1990

[1]Computer Science Dept., Stanford University,
Stanford, CA 94305
[2]Computer Science Dept., Brown University,
Providence, RI 02912
[3]Computer Science Dept., University of Milano,
Milano, Italy

Towards Megaprogramming

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

October 17, 1990

Abstract

Our goal in this paper is to develop an architecture and methodology for megaprogramming. We define *megaprogramming* as programming with very large modules called *megamodules* that capture the functionality of systems such as these used by banks, airline reservation systems, and city transportation systems. Megamodules are internally homogeneous, independently maintained systems whose life cycle is managed by a software community with its own goals, knowledge base, and programming traditions. We use the term "ontology" to denote the declared entities that may meaningfully be used within a megamodude and their internal operations. A problem to be dealt with is the interaction of ontologies.

Computations using megamodules are described by *megaprograms*, written in a *megaprogramming language*. We are concerned with interfaces of individual megamodules and with the megaprogramming language for megamodule management. Each megamodule must describe its externally accessible data structures and operations; descriptions are stored in libraries, called megamodule repositories, that have certain data dictionary facilities. Megaprogramming languages provide more flexible module management than traditional methods of combining modules, and in particular may perform transduction between modules on a variety of computer systems. Modules will have different data representations.

Databases may be supported within megamodules with complete autonomy. They are not directly accessed by megaprograms. Some megamodules may share databases and use their facilities to resolve conflicts due to concurrent access. Input-output for human clients is generally performed by specialized modules.

Megaprogramming allows us to think in terms of larger abstractions than traditional programming, and is a form of component-based programming in the large. It has the potential of substantially increasing modeling power and software productivity because its powerful encapsulation allows very large software systems to be managed as collections of megamodules. Programming within a megamodule can be handled by traditional techniques for programming, while interaction of megamodules is governed by megaprogramming protocols.

[1] Computer Science Department, Stanford University

[2] Computer Science Department, Brown University

[3] Computer Science Department, University of Milano

1. Introduction

The term *megaprogramming* was introduced by DARPA [BoSc90] to motivate research on extending the modeling power and programmer productivity of component-based software technology. We develop a specific megaprogramming architecture and software methodology. Megaprogramming is a form of component-based programming in the large that must handle the following kinds of largeness:

- largeness in size of programs: many lines of code
- long extent in time: persistence
- diversity of concepts and people: software communities, knowledge, traditions
- substantial infrastructure: networks, tools, interfaces, education

Traditional programming in the large manages large programs by traditional techniques of modularity, but does not address the requirements of persistence, diversity, and infrastructure. Persistence in time requires greater attention to the management of change. Diversity of concepts and people requires the coordination of diverse languages, multiple programming paradigms, and heterogeneous components. Substantial infrastructure requires powerful tools and programming environments, as well as support of networks, system evolution, and education. We propose a paradigm for megaprogramming based on partitioning of tasks into large components called *megamodules* that provide greater spatial, temporal, and conceptual independence than traditional modules.

The remainder of section 1 introduces basic concepts and a running example of intercity transportation. Section 2 develops the concept of "ontologies" that determine the local terminology of each megamodule, and focuses on the specification and interaction of megamodules and megaprograms with different ontologies. Section 3 outlines the general features and functionality of a megaprogramming language. We define operations for supplying and extracting information, for invoking megamodules, and for inspecting their state and monitoring their progress. Section 4 deals with megaprogramming system architecture, with libraries and databases, and with megamodule maintenance and life-cycle management. Section 5 extends the example domain of the running example to a larger set of megamodules. Section 6 positions megaprogramming relative to other work.

1.1 Component-Based Software Technology

Software technology is making a transition from programs as action sequences to programs

as collections of interacting software components. This transition from action-oriented to component-based software technology is reflected in ADA, which supports a rich variety of software components, and by object-oriented languages, which focus on a particular kind of software component (data abstractions) as a basis for software construction [Wegn:90A].

The essence of component-based software technology is abstraction implemented by encapsulation. Each software component is an abstraction that encapsulates implementation decisions. Components provide their clients with an abstract, representation-independent interface that determines behavior. Encapsulation facilitates independence of development and maintenance as well as information hiding. Megamodules also provide information about themselves to help the megaprogrammer control the interaction with the megaprogram.

Components may be classified by the kinds of information that they can encapsulate:

functions and procedures: encapsulate statements (action sequences)

data abstractions: encapsulate data (state specification)

megamodules: encapsulate data, behavior, knowledge, concurrency.

The earliest software components, functions and procedures, encapsulated action sequences, namely statements and expressions. Data abstraction improves upon procedure-oriented abstraction by encapsulating data, but data abstractions are generally restricted to encapsulate only data variables and values, and not type, class, and knowledge specifications. Component-based software technology will evolve by supplementing data abstractions by more powerful abstractions that hide local knowledge and behavior as well as local data, and this ability will allow more powerful abstractions to be specified.

In object-oriented systems, knowledge and behavior are specified by classes. In most object-oriented systems, classes are accessible to all objects and are therefore not encapsulated in any particular object. Global accessibility of classes results in behavioral uniformity, since all software components have access to all behavior specifications. Global accessibility of classes thus enhances their reusability but constrains the diversity of real-world software components. Organizational diversity requires local encapsulation of data, knowledge, and behavior. These aspects are inherently local to the software community in which a software module is developed and thus software components should encapsulate such features.

1.2 Megaprogramming

Megaprogramming strengthens the abstraction capability of components by allowing behav-

ior and knowledge as well as data to be encapsulated [Wegn:90B]. Software components that encapsulate behavior are called *megamodules* and programs that manage megamodules are called *megaprograms.*

Megamodules capture the interaction of organizations like banks and construction companies rather than that of objects with smaller granularity like bank accounts and bricks. They model networks of communicating workstations in disparate computational environments that cooperate on substantive common tasks. Megamodules can also represent non-physical systems like the transportation system of the city of San Diego or a world-wide airline reservation system.

Strong encapsulation restricts reusability across megamodules, but this simply reflects reality. Megamodules mirror real-world organizations and organisms in which internal subsystems (the human kidney or a car exhaust system) sacrifice reusability to realize efficiency.

Cleanly separating the concerns of individual megamodules facilitates their independent development and maintenance. Megamodules are like nation states with their own languages, traditions, cultures, and *nationalistic* loyalties. Once created, they have a dynamic of their own and cannot be eliminated without substantial social consequences. The social dynamics of megamodules should be explicitly considered in planning their life cycle.

Megaprogramming focuses on large abstractions implemented by strong encapsulation. It is a form of *component-based programming in the large*, while traditional procedure-oriented or object-based programming is by contrast a form of *component-based programming in the small.* Because of its stronger abstraction mechanisms, megaprogramming could yield a breakthrough in modeling power and software productivity that augments the human intellect and greatly increases the expressive power of programmers.

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 can be modeled by a megamodule, and intercity transportation by air, rail, and truck can be modeled by megamodules. The complexity of each megamodule is substantial and the total number of megamodules can be large. We will examine in greater detail a specific subproblem of this general problem.

Consider a logistics application for the shipment of goods between two 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, on Point Loma to Lindbergh Field

2 Commercial air freight

3 Surface transport in Washington DC from one of its airports to NRL in Anacostia

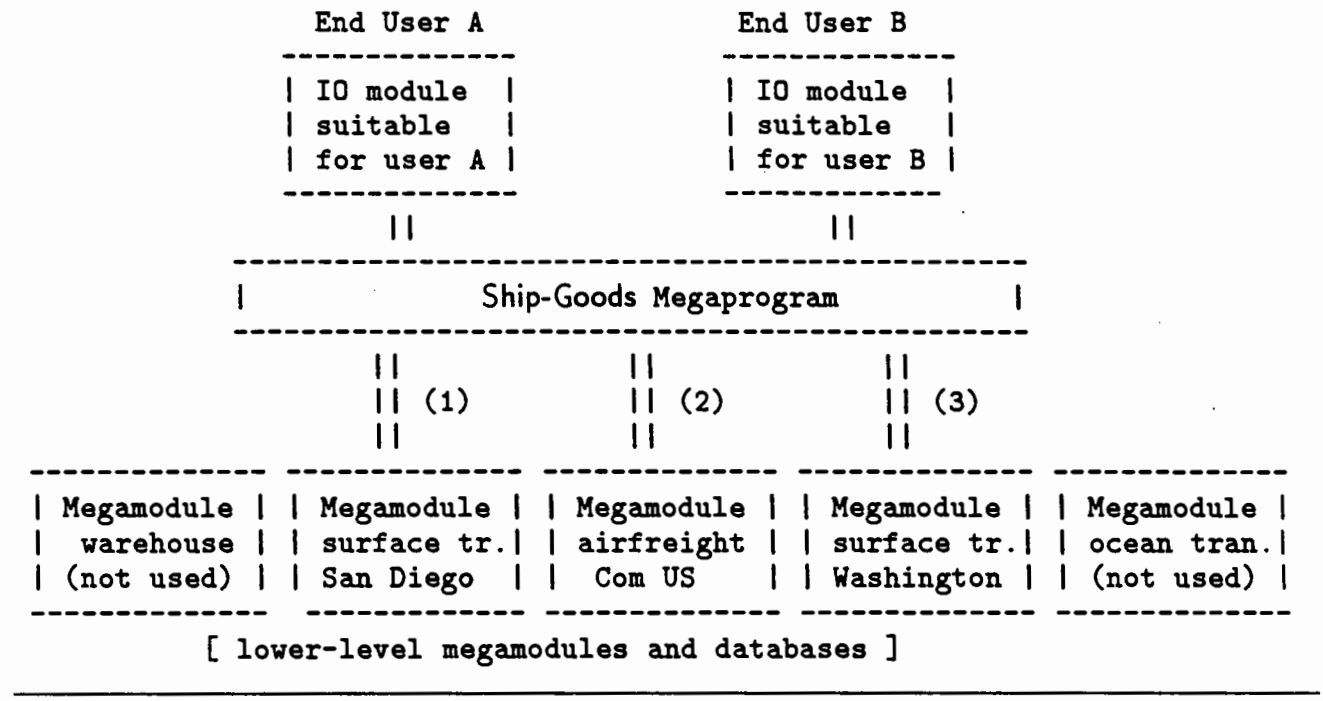

Figure 1. User view of megaprogammed architecture.

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, since it would be awkward for the San Diego expert to update the Washington databases with alternate routings if a road in Anacostia is blocked for some time due to repairs. Megamodule (2) will differ in ontology as well; since it represents and processes data according to its particular needs.

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 multiple megamodules. Since distinct airline databases represent their data differently, the module will also have to deal with differences and partial overlaps in formats, encodings, and semantics of those codes [DeMi:89].

The Washington surface-transport megamodule 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. It is not clear whether a single class of surface-transport megamodules can be defined for all or many cities (and regions) or whether local transport conditions differ sufficiently to warrant multiple classes.

1.4 Megaprograms and Megaprogramming Languages

Since megamodules are large, more data may be supplied and extracted from a megamodule than from a traditional module. Supplying information, extracting information, and invocation are therefore three distinct operations on megamodules.

Megaprograms coordinate megamodule computations by commands in a *megaprogramming language*. The execution sequence of megamodules may be affected by a variety of application requirements and constraints. If, in the *Ship Goods* example, the *time-of-arrival* at NRL is the critical factor, then execution must 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 should perhaps focus first on the most economical flight. Megamodules should be able to provide general services regardless of the order in which they are requested. Such knowledge will be dealt with by the megaprogrammer, possibly using special-purpose optimizers which can exploit flexibility in the execution of megaprograms. These considerations will affect the design of the megaprogramming language. Typically

flexibility is achieved by moving to a declarative language format.

Megamodules specify information that must be supplied prior to invocation and the information that may be extracted by clients after execution of the megamodule. For instance, the air freight module will have an externally accessible data structure that permits it to be supplied with, among other, the origin and destination cities. The elements to be supplied to the megamodule can either be placed into an accessible work area prior to execution, or more likely, shipped in a message to the computer where that megamodule resides. Similarly, the results extracted from the megamodules, namely estimates of transport task costs and time, can be extracted by the megaprogram in an accessible work area, that may be represented with a suitable database format.

Megamodules need not have direct interfaces for human use. The only requirement is that all data elements useful to an external user should be made available for extraction from the megamodule into the user interface environment. Since user interfaces include a wide range of equipment and locations, it is best to have specialized input/output software. This removes a heavy maintenance burden from megamodules and megaprograms, since external requirements change frequently. Megaprogramming concepts for *inter-systems cooperation*, may be considered separately from those affecting the user interface.

The megamodules of a megaprogramming systems are specified in a library or data dictionary. The specification of computations on multiple megamodules by megaprograms involves centralized coordination that mirrors real-world problem solving in large organizations. The high-level decision maker is aware of the capabilities of all system components in planning strategy for handling specific tasks.

1.5 Ontologies

Megamodules have internally consistent declaration, representation, interpretation, and type systems that reflect a uniform conceptual framework and problem-solving paradigm. But different megamodules may have different representation systems, since they model different knowledge domains and are developed for different hardware systems by different software communities. The representation and interpretation paradigm of an individual megamodule will be called its *ontology*. Ontologies provide a conceptual framework, both for talking about an application domain and as an implementation framework for problem solving. They provide a partitioning of context, so that systems which would be otherwise too complex to

understand, become manageable [WRB+:90] [GuLe:90]. Ontologies model the distinctions of knowledge in systems whose main components differ widely in their goals and purpose. We focus on the interaction of megamodules with different ontologies and the construction of megaprograms that manage megamodules with multiple ontologies. The specification of ontologies local to a component remains a local issue.

1.6 Research Issues in Megaprogramming

Megaprogramming will differ qualitatively from conventional programming. The following questions will need to be addressed:

- How can we encapsulate behavior, knowledge, and concurrency?
- What is the role of megaprograms in coordinating megamodule computations?
- How should megamodules communicate with one another and with users?
- What data-conversion facilities are needed for megamodule communication?
- How should megamodules be developed, maintained, and enhanced?
- How should megaprograms cope with changes of constituent megamodules?
- How should megalibraries consisting of megamodules be organized?
- What global assumptions (protocols) should govern megamodule interaction?

In the remaining sections we develop a framework for addressing these research issues.

2. Ontologies: Their Specification and Interaction

Megaprogramming systems must support the interaction of megamodules with radically different local declarations and data representations. To provide a framework for modeling such systems we borrow the term "ontology" from philosophy and artificial intelligence, where it denotes the primitive concepts and terminology of a domain of discourse [Hay85]. The ontology of a megamodule determines the representation and properties of the collection of declared entities accessible within the megamodule. Megaprogramming recognizes that the specification of ontologies is specific to individual megamodules, but that we have to enable communication among subsystems having different ontologies.

The meaningful entities within a megamodule are defined by local or global declarations. 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

data declarations for variables, operation declarations that determine rules of computation, and knowledge specifications. In increasing order of level of committment we can list

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

Variables and operations may be introduced by traditional declarations, while knowledge is more difficult to specify and to integrate with other declarations. The challenge is to find ways of representing relevant knowledge, such as the knowledge of time, by constraints that are consistent and usable with traditional declarative specifications.

Ontological commitments for knowledge are more complex than for data and operations. In our model, ontological commitments for knowledge are specified and implemented with megamodules. We think of megamodules as knowledge specifications, and implement an ontology of time as a megamodule whose interface specifies its ontological commitment. The megamodules for local transportation in San Diego and for commercial air freight are knowledge specifications that make ontological commitments to clients megaprograms. But we may wish to implement ontological commitments to knowledge with less overhead than that incurred by a megamodule.

Megamodules encapsulate local definitions of data, behavior, and knowledge. They operate in an environment that may specify additional data, knowledge, and behavior. The ontology of a megamodule specifies local and global data, knowledge, and behavior in a form that allows us to reason about and perform computations on its domain of discourse. The following questions must be addressed:

1 How can data and behavior declarations be extended to knowledge declarations?

2 How should encapsulation of data be extended to behavior and knowledge?

3 How can knowledge (of time, space, money) be effectively specified?

4 How should megamodules with different ontologies interact?

Megaprogramming systems recognize an internal ontology for each megamodule, as well as a global ontology that knows about megamodule interfaces specified in a library or class dictionary, and possibly other ontologies corresponding to "views" of a megamodule or subset of megamodules. The ontology of a megaprogram subsumes the interface ontologies for all accessible megamodules. Communication among megamodules is handled by a megapro-

gramming language that handles conversion of data from the representation required by one ontology to that required by another.

Ontological commitments should be *consistent* in the sense that the various named entities in any environment can be used without fear of conflict or inappropriate usage. Such consistency is required within a megamodule, during communication among megamodules, and in the megaprogramming environment. Since the number of megamodules can be large, the question of checking consistency can be acute, and we must try to define as precisely as possible what we mean by consistency. The criterion of type consistency is well-understood for typed languages. Research is needed to determine whether the notion of type consistency can be extended to cover ontological consistency of megaprograms or whether a new notion of ontological consistency will have to be developed.

We are also interested in the adequacy of ontologies for capturing the phenomena in their domain of discourse. If we have an independent specification of what the ontology should model, then we can determine whether the ontology meets this specification.

The notions of consistency and adequacy introduced above relate to the logical notions of soundness and completeness. Ideally, ontologies should be both sound and complete in precisely capturing the desired phenomena of a domain of application. The problems of demonstrating ontologies sound and complete are closely related to showing that megamodules meet their specifications. Formal specification of megamodules presents greater difficulty than formal specification of traditional models. Representative recent research in this area may be found in [LeTr90].

Data abstractions have a simple, but not trivial, ontology, that can serve as a starting point for specifying more complex ontologies. Their encapsulated declarations may include state variables (instance variables) and a consistent set of operations, some of which have both an external interface to clients and an internal interface to local instance variables. Type specifications permit the consistency of external and internal interfaces to be checked. Concurrent data abstractions include consistency constraints on the accessibility of shared resources.

Megamodules have a more complex internal structure than data abstractions and therefore a more complex ontology. The consistency requirements must be extended from type consistency of interfaces to the consistency of knowledge and behavior. Ontological commit-

ments for notions such as time and space are required to be consistent with other ontological commitments in any environment of use [Shoham:88].

Ontological commitments of a megamodule can be hidden. A megamodule which understands time internally can produce time-independent results which can be combined with results extracted from other megamodules that do not understand time.

The ideas of declarative power, encapsulation power, and subsystems with different declarative structure play a key role in our megaprogramming model. These concepts could in principle be developed without introducing the term "ontology". However, this term is useful in focusing on the specification and interaction of heterogeneous encapsulated software components. It plays a pivotal role in modeling the distribution and accessibility of knowledge in systems whose components (subsystems) can differ widely in their level of knowledge and specialization.

Our concern with interaction among different ontologies goes beyond work in philosophy and artificial intelligence on specific single ontologies. Moreover, computational ontologies are more demanding in their requirements of computational robustness. We recognize that much research is needed before practical megaprogramming systems based on these ideas can be implemented.

2.1 Example: A Payroll Ontology

A payroll ontology must define the concepts and terminology needed to specify and perform payroll computation. This includes a data structure for employees with their working time, benefits, and deductions. Rules, typically incorporated in programs, define accumulation of vacation, taxes withheld, benefits, and, via some computation, a measure of employee cost. The programs used within the payroll module are peculiar to that data representation. Other payroll programs may have a similar ontology.

Non-US employees fall outside this payroll ontology, as do contract personnel not reimbursed through payroll. Megaprograms may have to invoke megamodules for these categories as well, and combine their results. Integration of all these megamodules into a single megamodule is not desirable. Indeed, a megamodule for German payrolls is best maintained by a German expert, rather than by a US programmer, who is then expected to decipher German tax rules and apply them according to the German tax year schedule. In today's programming style, integrated programs require costly integrated maintenance.

A megaprogram can extract some of the results of the payroll megamodule to calculate the total cost of a major project and can use the results without worrying about the internals of payroll computations. If an approximate estimate of project costs is sufficient, then the cost of its invocation should be correspondingly small. Results may be cached by the megamodule for improving the performance of future references, or the computation may be repeated whenever requested [Rous:86].

More importantly, the payroll megamodule can be maintained independently, and changes in taxation rules do not affect the structure of the megaprogram. Furthermore, the maintainer of the payroll megamodule need not access the megaprograms that incorporate the payroll megamodule to validate changes.

3. Megaprogramming Languages

Megaprograms are written in a megaprogramming language MPL that invokes megamodules and organizes communication among them. They can invoke megamodules, extract data from megamodules, perform a transduction that transforms extracted data to the representation required by a second megamodule, and supply the transformed data to that megamodule.

To illustrate MPL, we give a 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
  SUPPLY TO Producer, list-of-parts
  INVOKE Producer.product
  EXTRACT FROM Producer, list-of-produced-items
  PERFORM Transduce(list-of-produced-items, list-of-consumer-items)
  SUPPLY TO Consumer, list-of-consumer-items
  INVOKE Consumer.purchase
  EXTRACT FROM Consumer, list-of-purchased-items
  Output cost, list-of-purchased-items
```

Figure 2. A Simple Megaprogram

SUPPLY and EXTRACT are used to exchange information between a megaprogram and a megamodule; INVOKE is a megaprogramming statement that causes the actual execution of the megamodule function. A number of additional MPL statements, as IMPORT SUPPLY, IMPORT EXTRACT, INSPECT, CONSTRAIN, EXAMINE, and ESTIMATE, are used during the development of a megaprogram and during megaprogram execution to extract meta-information from the megamodules.

MPL is a relatively low-level language from the programming language point of view, but is high-level when viewed as a networking language that transmits complete formatted data structures between megamodules at different nodes of a network. MPL provides great flexibility in mediating communication among megamodules but may, for simple patterns of communication, be compiled into more efficient constructs. After presenting MPL, we examine the tradeoffs between flexibility and efficiency, explore higher-level megaprogramming languages, and consider megamodule composition.

3.1 Specific Language Commands

Figure 3 contains a partial list of the megaprogramming language commands of MPL. We classify the language commands into the following classes:

information transfer commands

invocation commands

monitoring and inspection commands

constraint specification commands

computation commands

control commands

type commands

concurrency control commands

Information transfer commands are needed before invocation to supply megamodules with data and after invocation to extract results. Invocation commands are central to performing the computation, but may require preparation and coordination. Monitoring and inspection commands are needed to keep track of the progress of the computation, and can sometimes be used to change the course of the computation. The circumstances under which invocation takes place may be constrained by constraint-specification commands. Computation commands are used for transduction and general computation. Control commands

specify conditional execution and loops. Type-specification commands are required for type consistency and data conversion. Synchronization control commands are required because megaprogram computations are highly concurrent.

```
-- INFORMATION TRANSFER COMMANDS
      PRESENT       output-module datastructure
      OBTAIN        input-module  datastructure
      SUPPLY TO     megamodule datastructure [PARAMETER parameter-name]
      EXTRACT FROM megamodule datastructure
-- MONITORING AND DISPLAY COMMANDS
      IMPORT SUPPLY  megamodule declarations
      IMPORT EXTRACT megamodule declarations
      EXAMINE        megamodule status-variable
      ESTIMATE       megamodule estimates
      INSPECT        megamodule datastructure
-- INVOCATION COMMANDS
      INVOKE         megamodule  [WITH parameters]
      PERFORM        megaprogram (parameters)
-- CONSTRAINT SPECIFICATION COMMANDS
      CONSTRAIN     megamodule
            FOR datastructure  APPEND           constraint-data /
                               OMIT             constraint-data /
                               REPLACE WITH     constraint-data
      LIMIT megamodule TIME (interval-length)
      LIMIT megamodule COST (time) / PRICE (amount)
      LIMIT megamodule RESULT extract structure TO  [BEST] (integer)
      ABORT megamodule
      TIME-OUT time-interval
-- COMPUTATION COMMANDS
      SET   <assignment>
-- CONTROL COMMANDS
      DO FOR <datastructure>          <<follow Ada syntax>>
      BEGIN block
      END block
      IF  condition  THEN  block  -- incuding megamodule STATEs
-- TYPE COMMANDS
      GET type specification
      CAST type-spec INTO type-spec FOR datastructure(element)
      -- other operations to combine types
-- CONCURRENCY CONTROL COMMANDS
      -- synchronization primitives
      EXAMINE megamodule status-variable AND WAIT IF ...
      -- transaction primitives as
      START, COMMIT, ABORT
```

Figure 3. Megaprogramming Language Statements

3.2 Data Structures

The *data structures* supported by a megaprogramming declaration conform to an MPL type system. The type system should be powerful enough to support complex data structures; therefore, it must include generalized type constructors, including records, sets, multisets, lists, sequences, insertable arrays, sparse arrays, etc.

Megamodules are maintained independently; the absence of conventional top-down control means that megamodules define their data structures asynchronously from megaprogram definitions. Megamodules can have arbitrary internal type sytems. Since megaprograms are defined after the megamodules are defined, the megamodules must accomodate a self-describing `EXPORT` type system when communicating with the megaprogams. This will allow the megamodules to export data structure declarations that are within the MPL typesystem. 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` structures.

The technology we propose to borrow and adapt here derives from database schemas, the description of database contents and formats that permit many transaction programs to share a database and obtain data selectively and associatively. Note that databases are also defined prior to their use by end-users' transactions.

Two megamodule functions, `EXPORT SUPPLY` and `EXPORT EXTRACT`, define the data structures to be used when data are to be supplied to and extracted by the megaprogram. These operations export data descriptions when executed by the megaprogram. They are executed in response to MPL commands, `IMPORT SUPPLY` and `IMPORT EXTRACT`. The MPL compiler will issue these `IMPORT` requests when `SUPPLY` data or `EXTRACT` data statements are encountered in the MPL program. The data structures have to be exported from the megamodule and imported into the megaprogram so that the proper code can be generated for subsequent data transmission. Data transmission code must be generated for supplying the megamodules prior to their execution, for transducing data among megamodules, and for extracting data for final presentation from megamodules.

A significant extension of database schema concepts will be required to support the concept of data definition `EXPORT` from megamodules and `IMPORT` onto megaprograms:

1 The data structures permissible in general programs are more complex than the simple relations seen in most DBMSs.

2 Enough information must be provided to permit transduction of information between data structures. Transduction occurs when data are submitted to a successor megamodule by the megaprogram using a SUPPLY statement; and those data are obtained from another megamodule, using the EXTRACT statement. We count on optimizations by the MPL compiler, in which consecutive EXTRACT and SUPPLY operations are combined to generate a direct flow of information among megamodules.

3 Authorization levels will have to be refined, as discussed below.

A number of other statements are defined to effect the operation of the megamodules or to provide information about them. We wish to give megaprograms much information, so that they can use these large modules effectively.

Figure 4 summarizes the data interactions between the megaprogram and a megamodule. Although the figure depicts the traditional precedence of compiling versus execution, the operations provided in MPL do not require such a strict ordering. Especially compiling can be performed incrementally, as more information about megamodule status becomes available.

We now discuss selected operations in MPL in more detail.

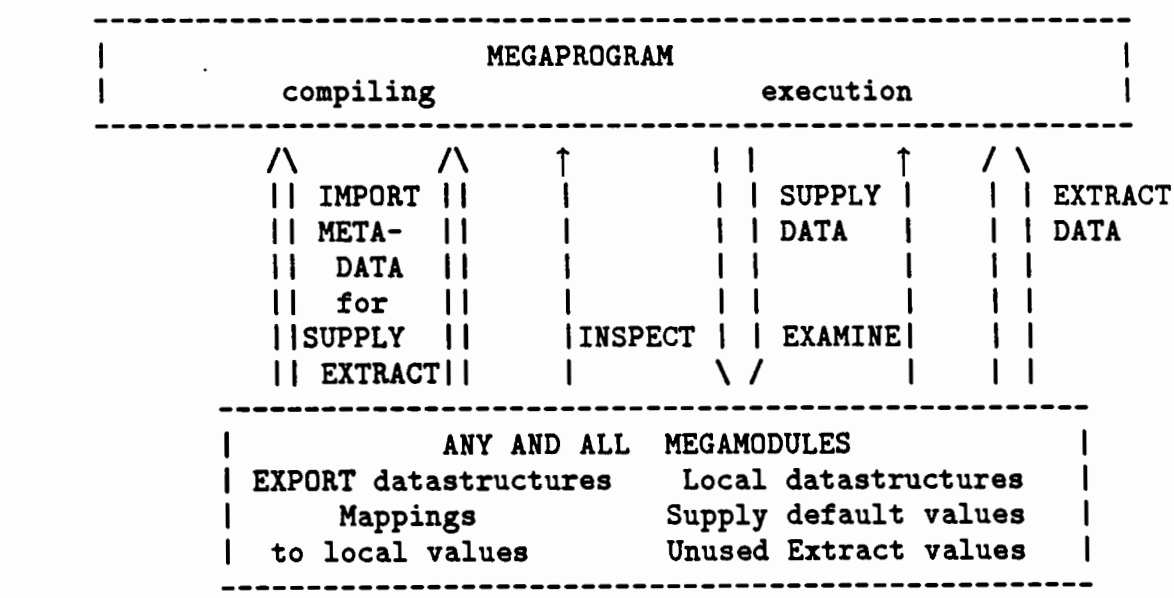

Figure 4. Information flow in megaprogamming.

3.3 Supply, Invoke, and Extract

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.

Supply The `SUPPLY` statement provides information to the megamodule, so that its invocation will work on the appropriate data. A `SUPPLY` statement is always associated with data structures defined in the MPL type system. When the supply statement is compiled for a specific megamodule, the type of the MPL data structure is compared with the type of the datastructure definition previously exported in response to a `IMPORT SUPPLY` command. The megamodule is responsible for managing data supplied to it in accordance with the specification it has exported. Any type coercion transformations needed for local compatibility with the megamodule type system are the responsibility of the megamodule. Problems of this translation inside a megamodule are not discussed in this document.

The supply data structure is therefore shared by all the invoking megaprograms. However, data values supplied by the various megaprograms conforming to this schema are different. In particular:

- Many values may not be supplied by the megaprogram; default values are assumed by megamodules for supply variables that have actually not been supplied.
- Some elements of the `SUPPLY` data structures may be defined as `PARAMETER`s. Parameter values are specified when the actual invocation of the megamodule takes place.

Processing of a `SUPPLY` statement is likely to cause at least partial compilation of the megamodule. Compilation is possible because of the availability of the static binding of the supply data structure to values. Only variables explicitly predeclared to be `PARAMETER`s are not bound until the megamodule is invoked.

In most cases, compilation generates a specific *executable code* for the specific megamodule in the specific megaprogram; input parameters are expected to be provided at execution by the `INVOKE` commands. However, other cases are also possible, as discussed below for the `INVOKE` statement.

During execution of the `SUPPLY` statement the megamodule retains control. This assures that the data transmission is synchronized. But arbitrary interactions can take place prior

to, and during the execution of the INVOKE statement.

Invoke The statement "INVOKE megamodule WITH parameters" causes the 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 SUPPLY statement. After the INVOKE statement is processed, control is returned to the megaprogram, in the expectation that megamodule tasks are lengthy and the megaprogram will schedule other work in parallel. The EXAMINE statement provides for inpection of the status of a megamodule. For instance, the examination result "STATE='done'" indicates that the megaprogram can now extract results from the megamodule.

Note that INVOKE typically causes a local or remote procedure call, with the parameters of the invoke being part of the activation message; it can hovever be dealt in other ways. For instance, we may imagine that a megamodule is always active and keeps the EXTRACT data structure always up-to-date. Then, the INVOKE message really corresponds to no operation, and the EXTRACT operation may be immediately executed. Similarly, the megamodule might be active periodically, and keep versions in the EXTRACT data structures; once again, the EXTRACT operation may be immediately executed, since it is not going to affect the behavior of the megamodule.

The INVOKE statement is a useful primitive also in the case of interacting with independently executing megamodules. It defines the synchronization point and is associated with book-keeping activities and security checking.

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 EXTRACT operation fills a datastructure exported to the megamodule with values produced by the megamodule. The extract operation is compiled with respect to the EXPORT data structure, which defines the type of all the variables that may be exported from a megamodule as the result of its invocation. Most megaprograms will only extract some of the information made available by a megamodule.

The result is fully valid when the megamodule state is done, but we place no constraints on earlier extraction.

During execution of the EXTRACT statement the megamodule again retains control. This

assures that also in this phase the data transmission is synchronized.

3.4 Operations for obtaining the EXPORT information

The following operations are used in megaprogram creation to determine the data structures and defaults in the EXPORT information of a megamodule.

Import Supply The IMPORT SUPPLY statement causes the megamodule to export a listing of all data elements that must or can be supplied. Standard names are used to describe data structures, formats, extents, and sizes, following the MPL typestructure. We envisage a syntax and capabilities similar to ADA declarations or database schemas. The IMPORT SUPPLY statement is emitted by the MPL compiler when a SUPPLY statement is encountered, so that the type representations required by the megamodule can be constructed. The source of the actual data to be supplied is either the megaprogram, one of its input modules, or other megamodules, in any combination. The MPL type system provides the *lingua franca* among these sources and the data structure specified by the IMPORT SUPPLY statement.

All elements to be supplied will have defaults, that only specifically relevant information needs to be transmitted. An INSPECT command can retrieve the actual default values for any supply items.

Import Extract The IMPORT EXTRACT statement causes the megamodule to export a similar list as IMPORT SUPPLY for available output data. Again, the megaprogram compiler will emit this statement when an EXTRACT statement is encountered, so that it can compile effective format translations and transductions to other megamodules or to output modules, as specified in the megaprogram.

Rarely will all result variables from the IMPORT EXTRACT statement be actually extracted. Since extraction can occur at any time after an INVOKE statement, a standard convention for results which are either null or not yet available will be needed.

Examine In addition to the data structures used to supply and extract data, a megamodule maintains a number of state variables. Some of these should also be accessible. The EXAMINE statement accesses the most critical variable, namely STATE. In megamodules STATE can have values as `done`, `in_progress`, `idle`, `in_error`, or `waiting`. Other variables that may be examined are measures of nearness to a solution for an iterative program, the number of choices being considered for a search-type megamodule, etc.

These variables may be accessed at any time for a megamodule that has been invoked. Only valid values can be returned, and the operation of `EXAMINE` is synchronous with respect to the megaprogram. The megamodule has to process an `EXAMINE` request with little delay.

3.5 Operations for inspecting the megamodule status

A number of further operations permit interaction of megaprograms and megamodules. Their function is to aid in compilation and execution to attain a high level of system 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.

Inspect Inspection of a megamodule provides information about its ontology and specific parameters. The result of an `INSPECT` operation is specific to the megamodule being inspected. A candidate datastructure would be a hypertext representation. In the Washington surface transport example inspection could reveal that the algorithm must perform a search through a very large space. Inspection of the air-freight megamodule can also report the hubs airports considered in its heuristic. The results of inspection do not depend on the current use of the module being inspected.

Estimate For general megaprogramming, estimates of execution cost of megamodules may be needed. The result of execution of an `ESTIMATE` statement is simple: a value of the estimated execution cost (t time, d dollars) and the degree of certainty of that estimate (0...1). The value will differ depending on how well the current task is understood by the megamodule. For instance, the cost of executing the air-freight megamodule will be less when a constraint has reduced the number of hubs it has to consider. The uncertainty of the cost of obtaining the result also reduces as more constraints are given or the supplied parameters narrow the choices. Initially, if the module has not yet been supplied with specifications, the estimates will be quite uncertain, since the expected running time is poorly bound. Estimates may be explicitly obtained in the megaprogramming to help the megaprogrammer select megamodules or to embody scheduling alternatives into the program. They can also be used, invisibly to the megaprogrammer, by a megaprogram compiler to create an optimal program.

Constrain The `CONSTRAIN` statement permits resetting, for the specific application, of

internal parameters obtained by inspection. For instance, we may decide to limit the hubs being used by the air-freight flight scheduling heuristic.

Limit The `LIMIT` function constrains megamodule execution in terms of real-time intervals or cost (as a surrogate for computational time spent). A more novel alternative is to `LIMIT` a megamodule to only providing a small set of `BEST` results. Not all megamodules will be able to accomodate such a request, but if they can, the megaprogram task would be greatly simplified.

3.6 Authorization for Change

The megaprogramming concept depends on a partitioned assignment of responsibility to megaprogrammers (MPs) and megamodule programmers (MMPs).

A megaprogrammer (MP) will not be able to change the data structures provided by the megamodule. To some extent the structures are not visible to the MP, since the importation of datastructures for the `SUPPLY` and `EXTRACT` statements is handled by the MPL compiler. However, the `IMPORT SUPPLY/EXTRACT` statements do exist, and can be used directly by the MP as well. Rather than forbidding their direct use, we replace the concept of information hiding with the concept of change authority. Withholding such authority should be sufficient to protect the integrity of the interfaces.

In contrast, the programmers (MMP) that have the responsibility for a specific megamodule can freely change internals of their megamodule, but they do have a responsibilty to keep the `EXPORT` structure intact. Note that many megaprograms will use these data structures. Recompilation of a a megamodule should not alter the `EXPORT` structure either. If changes to the exported data declarations are needed, a new version of the megamodule is created, and it becomes the MMP's decision to move to the new module or stick with the old. In many cases megaprogram recompilation should suffice to move to a new version.

3.7 High-Level Megaprogramming Languages

The propsed MPL must be able to serve as a base for applications using higher level languages to communicate with their users. Suppose we want to deal with high-level commands like:

```
SHIP list-of-goods TO NRL
```

Such a command requires a `SUPPLIERS` megamodule that can find suppliers for each of the items in the list of goods to be shipped to NRL. Given the right kind of `SUPPLIERS` megamodule we could determine the suppliers by the following megaprogram commands:

```
SUPPLY TO Suppliers list-of-goods
INVOKE Suppliers.find-suppliers
EXTRACT FROM Suppliers list-of-(suppliers-goods)-pairs
```

We could then write a loop to compute the transportation schedule for shipping the goods to NRL.

```
FOR list-of-(suppliers-goods)-pairs
PERFORM Ship-goods (supplier, goods, NRL)
END
```

MPL allows each of the Ship-goods megaprograms in this loop to be executed asynchronously and independently.

Given the high-level specification, the megaprogram for shipping the goods could be written in MPL by a megaprogrammer. Alternatively, the translation from the high-level specification to a megaprogram could be performed by a megaprogram compiler. This would allow high-level decision-makers to directly communicate with the megaprogramming system. Thus a high-level megaprogramming language could be designed to allow executives and generals to directly specify a wide range of commands that would be understood by a compiler and compiled into a megaprogram.

Our example makes use of a SUPPLIERS megamodule and a parameterized Ship-goods megaprogram that takes the supplier, the goods to be shipped, and the destination as parameters. These parameters in turn determine the megamodules that will be involved in the execution of the megaprogram. A different city transportation megamodule will be involved for each of the different cities associated with suppliers.

Megaprogramming environments include a large number of megamodules, although a typical megaprogram is likely to use only about 7 ± 2 [Mill:56]. In the logistics case there will be at least one transportation megamodule for each city or region that has suppliers, and inter-city transportation megamodules for commercial air freight, trucks, trains, military transportation, etc. The city transportation megamodule might in turn be part of a cluster of megamodules for city services that include city schools, city garbage collection, city property description, etc.

Megaprogramming supports asynchronously executing megamodules, multiple concurrent processes within each megamodule, and concurrent execution within each megaprogram.

Concurrency control and synchronization is required to manage the many different forms of concurrency. The architecture of megaprogramming systems is further discussed in Section 4.

3.8 Composition and Interaction of 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.

Megamodules are more complex than traditional functions and procedures and in general require composition to be mediated by SUPPLY, EXTRACT, and Transduce operations. However, when the information flows and transductions are simple, the megaprogram is in effect unnecessary during execution and can be replaced by the MPL compiler with direct communication.

To elucidate these MPL concepts we can compare how direct communication among programs is specified in a UNIX environment. The MPL concepts we support 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 "$P1$ | $P2$" 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 $P1$. In UNIX direct communication is possible only in simple cases. In the proposed MPL pipelined execution is supported through compilation, which is to generate the equivalent of a pipe mechanism. Often the piping of a megaprogram will involve multiple processor nodes.

UNIX also provides simple transduction by filters that can be interposed in the pipe between two UNIX processes. The command "$P1$ | TRANS | $P2$" incrementally pipes the results of $P1$ to $P2$ via the transduction TRANS. In using MPL the TRANS code will be in the megaprogram, and can be compiled to be executed on the source, destination or any intermediary node.

In MPL, all megaprogram excution 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 comparison with UNIX brings out the generalizations provided by MPL. The generalization requires that we identify each MPL operation with an explicit megamodule name, since the compositions of IMPORT, SUPPLY, INVOKE, EXAMINE, EXTRACT, etc., are arbitrary and any of the megamodules may be addressed. Research will be needed to phrase a syntactically attractive language which permits specification of the parallelism and asynchrony implicit in the megaprogramming paradigm. Conventional languages focus on sequential composition. Megaprograms are best sketched as a directed graph, with cycles, that terminates to some goal state.

3.9 Composition and Interaction of Megamodules

Composition of megamodules is required in order to build and execute megaprograms.

A simple pattern of interaction between the megaprogram and megamodules, can be realized by SUPPLY, INVOKE, and EXTRACT. Note that even this simple pattern of interaction improves the conventional development of applications. The SUPPLY and EXTRACT data structures provide a higher level of abstraction with respect to direct linking of software modules, enabling transduction of data structures between the global and the local environments and differential compilation of megamodules depending on the needs of megaprograms.

More complex composition, yet still under the direct control of the megaprogrammer, is possible through the use of control structures IF ... THEN ... whose action is based on returned parameters from the INSPECT, IMPORT, EXAMINE, and ESTIMATE statements. By being able to INSPECT the behaviour 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 be then 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 requires a complex interaction when a megaprogram composition is in process.

Finally, we envision the possibility of *megaprogram optimization*, where the entire mega-

program is submitted to an optimization module. Such optimization would be responsible for:

- Combining two consecutive SUPPLY and EXTRACT operations by generating a direct flow of information between megamodules.
- Decide the order of invocation of megamodules, in particular invoking them in the "optimal" (e.g., fastest, less expensive, ...) order.
- Introduce parallelism and asynchrony in the schema of invocations of modules

These optimizations might seem inappropriate in the field of software engineering but have been successfully performed since a long time by database systems. Note that the format of information as provided by INSPECT, EXAMINE, ESTIMATE might be quite comparable to the information typically available to a query optimizer.

Megaprograms will normally operate on networks and invoke megamodules on multiple processors; these features will contribute to parallelism. Multiple megamodules can be initiated asynchronously and scheduled for execution. Megamodules permit such scheduling to be carried out at a high level of granularity, which is likely to be more effective than searching for fine-grained parallelism. The relative overhead of invoking a megamodule and communicating specifications and extraction request should be small. Within the megamodules, finer-grained parallelism is, of course, still possible.

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 Dallas airport, for the best flight, is now computed.

3 The Washington surface module is now invoked with that time and 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.

```
SUPPLY 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 5. Example of a megaprogram invoking one megamodule.

4. Megaprogramming System Architecture

A megaprogramming system consists of a collection of geographically distributed megamodules, and a megaprogramming environment that includes a dictionary of megamodule specifications. The system may have a variety of classes of clients that differ in the resources to which they have access. For example, clients may have a different level of security, may have access to only a subset of megamodules, or may be restricted to "views" that allow only a subset of megamodule properties to be accessed. The megaprogramming environment may be replicated, so that megaprograms may be written at many different sites.

The clients of a megamodule include not only megaprogrammers but also megamodule application programmers. Application programmers of a given megamodule must have the ability to directly access other megamodules. For example, the San Diego transportation megamodule may wish to ask the Commercial Air Freight megamodule for flight times in order to decide when it is appropriate for local transprotation to deliver goods to the airport. It is unreasonable for the megaprogram to foresee that the megamodule will require such information at the time of invocation, and provision must be made to allow the megamodule

to directly request information and computation from other megamodules. Communication among heterogeneous modules has been widely studied in MERCURY [LBGSW:88], and MACH [Spec:88], and the megaprogramming system must clearly support such communication.

The development and maintenance of each megamodule is entirely under the control 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. Megamodules typically have requirements for continuous execution of processes that monitor ongoing real-world activities, and can not be taken out of service because some of their subtasks may have critical real-time requirements, such as nuclear reactor monitoring.

A megaprogramming system is physically distributed and highly concurrent, supporting both intermodule concurrency and internal concurrency within a megamodule. Physical distribution has its logical analog in strong encapsulation that allows megamodules to encapsulate not only data, but also behavior, knowledge, and concurrency.

4.1 Megamodule Architecture: Multiple Interacting Ontologies

The ontology of a megamodule may include local and externally defined declarations. Local declarations may specify imported and exported data, they may be private declarations inaccessible outside the megamodule, or they may specify data that can be inspected but not modified. The set of invokable entry points determines still another category of locally defined information, corresponding to the interface of traditional modules such as objects.

Direct interaction of megamodules, outside of a hierarchical invocation to solve subproblems, is difficult to manage reliably and will have to be constrained. To avoid mixing ontologies we propose that any direct access of one megamodule by another be managed through ***megaviews***. Direct access rights may be more restrictive than access rights for megaprograms, and may differ for different clients of a megamodule.

A megaview has to resolve any ontological mismatch to allow a megamodule to directly harness knowledge encapsulated in another megamodule. For example, the San Diego local transportation megamodule could directly access the commercial air freight module to determine the best time for delivery of goods to the airport. But it could not deal with concepts of aircraft loading.

The variety of categories of declarative information within a megamodule is richer than that for traditional modules, as indicated in Figure 6.

```
MEGAMODULE name of megamodule
  IMPORT declarations for imported (supplied) data
  EXPORT declarations for exported (extracted) data
  PRIVATE local (hidden) declarations
  INSPECT data that can be inspected and monitored
  EXTERNAL externally defined, locally accessible declarations
  MEGAVIEW direct access to other megamodules (knowledge ontology)
  ENTRY entry points with parameters (traditional interface)
  implementation (body) of the megamodule
END name of megamodule
```

Figure 6. Declarative Structure of a Megamodule.

More research is needed to precisely define the sharability and accessibility of different kinds of declarations within a megamodule. The rules of distribution and accessibility for declarative information are a key to understanding the structure of megaprogramming systems [WWH+89]. Viewing large systems in terms of the scope and sharability of their declarative information provides valuable insights into their structure, and corresponds to viewing real-world systems in terms of the distribution and sharability of knowledge. Our notions of ontology and ontological commitment facilitate the understanding of systems from this point of view. The logical architecture of megaprogramming systems may be described in terms of the interaction of subsystems with different, but related, ontologies.

Local control of internal and interface declarations weakens the ability of system designers to enforce global system standards. But it reflects the division of responsibility in real software development, where local and interface declarations are entirely controlled by local software designers. Megaprogramming languages must be able to handle a variety of declarative interface specifications determined by developers of megamodules, although some general constraints on the form of declarative specification must be observed. The forms of declarative specification in a megaprogramming system, and the global constraints by which they are governed, is a matter for research, especially in the case of widely used concepts such as time, space, matter, etc.

4.2 Megamodule Repository and Dictionary

We expect that megamodules will be available through a repository. Such a repository pro-

vides linkages to people actively maintaining and benefitting from the utilization of megamodules. This distinguishes our vision from that of passive repositories, that often only contain abandoned software left at the end of a contract to fulfill documentation regulations. To find and exploit megamodules the repository must provide some facilities.

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. The type of information needed to be supplied and available for extraction, placed into some kind of a semantic hierarchy.
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 possibly cost of 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`, `EXTRACT`, `INSPECT`, and `EXAMINE` statements in MPL may be considered as formal interfaces to this database. These provide the specific, current detail for the information indicated, for instance, in points 3. and 4. above. The dictionary information is in general organized to enhance locating megamodules, rather than dealing with their specific execution.

4.3 Libraries of Megamodules

The data dictionary is effectively a "library" specification that specifies library resources available to megaprograms. A megaprogramming system has two levels of library facilities:

1. A distributed library of megamodules accessible to megaprograms.
2. Local libraries within each megamodule.

This corresponds to two levels of reusability: local and global reusability. Megamodules

sacrifice global reusability of local resources to facilitate local efficiency and local independence. The loss of reusability is an inevitable consequence of encapsulation. It occurs in data abstractions with local operations, which are not reusable because they are dependent on local instance variables for communication with other local operations. This reflects the specialization of components of real-world organizations and organisms, such as the payroll operation of a large organization, or the human heart, which have numerous interconnections to local subsystems and therefore can not be reused externally.

Libraries are repositories of software components that serve as reusable building blocks for software development. Megaprogramming languages serve as glue for composing megamodules, just as traditional programming languages serve as *glue* for composing traditional library components. Component-based software technology aims to construct programs almost entirely out of predefined components, with minimal use of glue. This generally requires domain-specific components tailored to a specific project or application. The ideal is to find for each application the *joints* that allow components to be designed for assembly into products with minimal glue. However, some applications are inherently more decomposable than others, and there is no guarantee that such joints will necessarily exist.

The paradigm for constructing programs from software components is not manufacturing large numbers of identical components but rather engineering customized products out of prefabricated components. The analogy of a plumber or a constructor of prefabricated houses is more appropriate than that of an assembly line. But prefabricated units have standardized interfaces, and when they have to be adapted to non-standard spaces, filler and plumbing extensions are required. Sometimes the process of configuring systems from components can be automated, as in automated computer configuration programs like XCON. But for one-shot applications the processes of reusing library components, designing new components, and configuring the system are intermixed. The megaprogramming concepts provide the IMPORT capability to specify those interfaces so that those fillers can be built.

Libraries in procedure-oriented languages have actions (procedures) as their software components. Procedure-oriented programs usually have liberal doses of glue in the form of statements of an underlying programming language. The seamless composability of procedures with programming language statements reduces the incentive for "pure" composition of procedural components, since the glue blends in so naturally with procedural components.

With megamodules, the glue is primarily SUPPLY, EXTRACT, and Transduce operations, but the megaprogramming language permits arbitrary megaprogram computations.

Megamodules may differ from traditional library components in a variety of ways. They offer a greater variety of services than traditional components. Different clients may have different modes of access to a megamodule, corresponding to different views. Megamodules are active components with independent execution mechanisms and an ability to dynamically control the mode of access by clients to their interface. They have a will of their own, controlled not only by the code of the megamodule, but also by programmers who can shut off access during maintenance or a hardware failure. Communication with a megamodule may require negotiation rather than simple communication.

4.4 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 SUPPLY 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 within a single megamodule.
6. Handle concurrent execution of multiple megaprograms initiated at different sites of the megaprogramming system.
7. Deal with the notion of transaction atomicity in the context of megaprograms.

4.5 Access to databases

We assume that databases are part of megamodules. In particular, the semantics of the database system is completely captured within the "ontology" of the megamodule and does

not need to become "public." Datastructures for EXPORT/IMPORT do become public. They will in turn be influenced by the schemas of databases which are stored within a megamodule.

We exclude the case of fully transparent distributed databases where access to any data item can be made from any site and actually regardless of the logical fragmentation and physical distribution; this issue is not the concern of megaprogramming. Our view is consistent with autonomous and independent access; however, we are can lower requirements with respect to federated databases [Litw:86] because we do not expect to provide comparable semantics across databases which are managed within different megamodules.

4.5.1 Concurrency Control

Databases maintain the persistent representation of the real-world. It is there where concurrency control is needed when megaprograms share the same databases, possibly via distinct megamodules. We may safely assume that durability (e.g., permanence of effects of committed transactions), local serializability (i.e., equivalence of each local schedule to some serial 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 practical solutions have been studied; these include *nested* and *long-lived transactions*. With nested transactions, subtransactions can locally commit even if the global transaction abort, as their actions can be undone by compensation (typically, by firing another transaction that will reconstruct the initial state). Long-lived transactions typically release their locks and run at a lower level of consistency with respect to full serializability.

When the limitations of nested or long-lived transactions are not acceptable, then the problems due to megaprogramming update to databases are eased by letting each megaprogram only update the database of one megamodule; this restriction is typically accepted at the current state of database technology.

4.6 Maintenance

Maintenance is greatly facilitated by the separation between megamodules and megaprograms. Maintenance takes place at the two levels separately.

- Maintenance within a megamodule is done by an expert of the megamodule's "ontology." We distinguish two types of changes. *Local changes* do not affect the EXPORT and IMPORT data structures, and therefore do not require megaprogram recompila-

tion. *Global changes* affect the EXPORT and IMPORT data structures, and therefore cause megaprogram recompilation. Such recompilation will be succesful when all variables appearing in the old versions of EXPORT and IMPORT data structures are present in the new versions.

- 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 across 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.

In summary, the rules for maintenance differ when megaprogramming:

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

Megaprogramming: A megamodule programmer (MMP) can only touch a single Megamodule.

For maintenance the megamodule maintainers will use their own program. This program is identical in form to any megaprogram, but likely to access only one or a few megamodules. The maintainer will, of course, have greater access privileges, specifically update privileges on internal structures. The maintainer may also request recompilation, to effect a new binding and keep the megamodule efficient.

5. Composing Megaprograms

The benefit of megaprogramming comes about when megamodules are used in a variety of megaprograms. We will illustrate our vision of shared use of megamodules by extending our example domain.

In Figure 7, we show several megaprograms which exploit different configurations of megamodules. Such megaprograms, beyond the Navy shipping megaprogram for NOSC-to-NRL, using the three megamodules for local and air transport, that we can hypothesize are as follows:

Goods Tranfer The shipping megaprogram may be subsumed by a more general transfer program which includes megamodules dealing with warehousing operations at the two

terminal cities. The detailed specification of the goods will now be extracted out of the delivery warehouse and supplied indirectly to the surface transport module in San Diego. The receiving warehouse in Washington has to allocate space and manpower at the time of receipt.

The cost calculations may invoke yet other modules, as a Payroll megamodule which permits estimation of employee cost.

General shipping A more general logistics megaprogram will also be able to invoke surface transport modules in many other cities. Now the number of potential megamodules for all locales will be large. It becomes reasonable to only those modules that are candidates for a particular problem. Late binding for the locales is appropriate now. Since all `surface-transport` megamodules will have similar ontologies, the cost of late binding is modest. We show later a feature of the proposed megaprogramming language which permits selected input to be designated as *parameters* for late binding.

Those modules may share code as well, but their databases will differ. To the megaprogram code sharing among megamodules is immaterial. Highly active megamodules may in fact be replicated on multiple processors. Issues of concurrency come into play when multiple megamodules access the same databases, say request simultaneously resources from the San Diego motor pool.

If we must ship the goods to locales without local facilities owned by the company, a general megaprogram may have to invoke a megamodule which will schedule local contractors for surface transport at those locales. If no contractor modules exist for those locales, yet other megamodules, which perhaps arrange for simultaneous shipment of trucks and personnel, will have to come into play.

Commercial Shipping The commercial air freight module will, of course, not only be usable by the Navy, say, but by other companies, say General Motors, as well, although these companies may impose different conditions and resources on surface transport. Toyota will have to consider sea shipment for imported parts as well.

Logistics A more general military logistics megaprogram will invoke megamodules to evaluate military transport options as well. The megaprogram will weigh the relative results extracted from those modules, depending on the policies of performing transport tasks by commercial contractor or in-house. These policies can differ depending on the goods shipped.

Evaluation of its own execution is not considered within the megamodules, since only the megaprogram has the global application view.

As we developed our example, we implicated further megamodules, such as warehouse modules, weather modules, etc.. These can be used in different configurations by a variety of megaprograms. Figure 7 sketches the configurations mentioned, and their `megamodules` which form megaprograms.

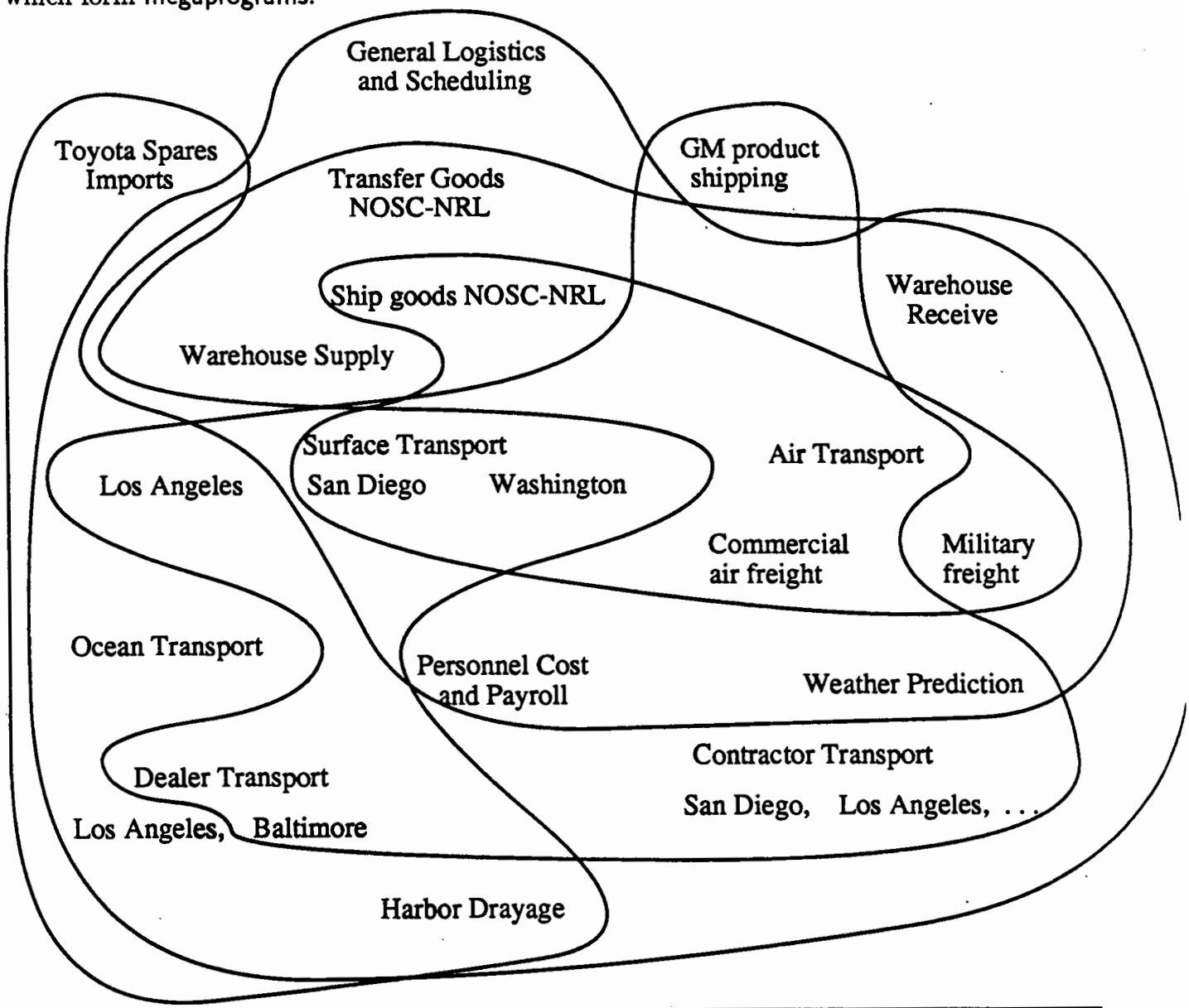

Figure 7. Megamodules in various megaprogram configurations.

Figure 7 does not show input-output or database modules. While at some level of abstraction they are equivalent to computational megamodules their higher degree of specialization permits us to focus on the specifics of this architecture.

5.1 Asynchrony of megaprogram execution

The megaprogramming paradigm is oriented towards parallel and asynchronous operation of the megamodules. Since megamodules will often reside on distinct and possibly remote computers, the ability to SUPPLY multiple modules in parallel, schedule their invocation independently, and EXAMINE their progress is important. The EXAMINE capability goes beyond testing if a megamodule has completed its task, which is sufficient for the traditional *Rendezvous* paradigm of parallel scheduling. The EXAMINE statement can also report the ongoing state in a megamodule. A module which appears not to make much progress, perhaps because it does not have enough data to constrain its search, can be abandoned and the results from another module, which may be simpler and produces results that are less optimal can be extracted.

The megaprogam for General Logistics, sketched above, may find that the `military transport` resources are stressed, so that it is costly to create a satisfactory schedule in that megamodule. The `commercial transport` megamodule, which has access to several commercial databases, could then produce a schedule more rapidly.

This example also points out an inherent weakness in this partitioned approach, which is due to the mutual independence of the megamodules. Since the megamodules do not interact directly, they cannot propose a schedule which uses both military and commercial transportation resources. A sophisticated megaprogram could, of course, investigate schedules where, on long-distance routing, likely transfer points (Frankfurt, Honolulu) are considered and the megamodules are again invoked with route-segments. Since the SUPPLY statements have already provided most of the data, subsequent invocations should be less costly to execute. In such case the terminal airports are best specified as late binding PARAMETERS.

5.2 Megamodule lifetimes

We see that these megamodules must have a lifetime of their own, overlapping the application lifetimes. Most of the modules will be maintained by specialists, not by the users. The life for a sharable module must now subsume the lifetime of all project tasks that invoke it. To make such a life feasible, and still permit orderly growth, version management for megamodules

becomes essential. A particular challenge here will be to identify the extent of differences among versions of a megamodule, to permit maximal adoption of the latest version.

We revise the traditional notions of life-cycle management from being application-based to being megamodule-based. In our capitalistic world there might well be charges for the use of the intelligence embodied in the megamodules, just as one is charged today for the use of some publicly available databases. These notions may not be easily accepted by the software industry and their programmers. Some programmers consider working on large applications a guarantee of employment for their lifetime.

5.3 Modules for interfaces for humans

The accessing megaprograms provide all of the required input-output, if need be through specialized input-output modules. Input-output modules can adapt to the wide variety of output modalities that are now available: print, X-windows displays, voice, etc. These modules must be supplied with descriptive information as well. Here the descriptions are used to map information for external presentation and label the output values. A nice example of the capabilities we would like to see in these modules is shown in [Cha:90] for menu input and in [McKi:86] for graphic output.

The argument for omission of input-output from computational megamodules is one of simplicity, shareability, and life-cycle management.

6. Comparison With Related Work

Since we are addressing a broad topic, the work of many computer scientists concerned with large scale systems is relevant to megaprogramming.

6.1 Software engineering principles

The role of software engineering in providing useful concepts for megaprogramming has been outlined earlier. Fundamental concepts include:

- Encapsulation and modularization principles, which are raised one level up by megaprogramming. These are fundamental programming concepts also inside megamodules.
- Software reusability, which is greatly enhanced by providing clean and general-purpose interfaces to megamodules through combined notions of `IMPORT - EXPORT` and `SUPPLY - EXTRACT`.

- Reverse engineering, which is needed in order to understand the ontology of existing modules and to derive the schema structures for both the SUPPLY and EXTRACT data structures.

In all these areas the megaprogramming paradigm provides a step forward.

6.2 Flexibility of composition

The need for flexibility in composition for megaprograms induces the requirement for a megaprogramming language. Simple, fixed composition, using CALL type sequences to subsidiary programs or the use of UNIX pipes does not provide for those rearrangements. The megaprogramming language interpreter must be able to generate alternate execution sequences for alternate conditions. These conditions will become more complex for more substantial megaprograms.

6.3 Independence of modules

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]. Since programming was not an objective of AI researchers, their focus has been on the intelligence within the modules and on loose control structures. The distinction in megaprogramming, as presented here, is that the control, even if complex, is directed towards solving the task of the megaprogram. We do not expect that megamodules propose and negotiate their own agendas, as is often the case in computational models for AI.

6.4 Transaction modularity in databases

Databases achieve task independence by breaking applications down into minuscule tasks, transactions, which only interact with each other through the database. Very many transactions comprise the workload of database systems, up to thousands per second. However it has become clear that not all database applications fit into this basic paradigm, so that related work is going on under the rubrics of *long transactions* [Katz:83] and *nested transactions* [Moss:85].

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].

The STRETCH Esprit project is considering the issue of providing a *Common object manager* (COM) software layer supporting heterogeneous language paradigms for databases (in particular, object-oriented and deductive); many of the concepts of megaprogramming may be useful for building the COM architecture, and some of the ideas in this report are actually borrowed from [Zica:90].

7. Conclusion

We have defined megaprogramming as a departure from standard top-down programming practice, which has not been able to deal well the problems encountered when building and maintaining large software systems [Paul:89].

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. The problems due to the large size and diversity of ontologies seen in megaprograms are being dealt with in a conventional scientific paradigm: *divide and conquer.*

The partitioning to achieve this division is achieved by defining megamodules, which show much more independence in terms of their structure and interfaces than traditional modules. They may be as large as conventional software technology permits, while retaining manageability and maintainability. We have defined megamodules as the unit of use and reuse by megaprograms in a manner that use and reuse become indistinguishable.

The combination of megamodules and megaprogramming is based on two simple principles, common to much of software engineering:

1 abstraction of complexity through modules: divide

2 hierarchical composition of those abstractions through megaprograms: synthesize.

The essential difference is that the abstractions in megamodules have a life independent of the megaprogram which uses them.

Both megamodules and traditional modules hide information to realize abstraction and restrict the generality of internal components to allow their efficient coordination to realize an external abstraction. But the nature of hidden information and of the restriction on

internal components differs for megamodules and traditional data abstractions. Since the megaprogrammer can not control or access internals of the the megamodules we need not to worry about misuse.

It is obvious that we are scaling up the role of a programmer when we move to megaprogramming. The essential differences between megaprogramming and programming of a megamodule are shown in the table below:

Activity	Programmer of a megamodule	Megaprogrammer using megamodules
Selection of functions	`Sort, sqrt, invert`	`Transport, Load`
Finding produced results	check documentation	via `IMPORT EXTRACT`
Finding needed parameters	check documentation	via `IMPORT SUPPLY`
Setting of parameters	set all needed for function	supply selected ones
Access databases	write embedded SQL	handled inside megamodules
Specify control flow	set counters, test conditions	set counters, test conditions
Interact with user	write i-o sequences	invoke i-o modules

This report has identified a number of open research issues:

- Understand deeply the nature of "ontologies" to the extent that they can be used for megaprogramming.
- Design a megaprogramming language and type system.
- Design and implement the megaprogramming support environment.
- Optimize megaprograms, which an increasing degree of sophistication.
- Understand and disclose the methodological issues concerned with building complex megaprogramming applications.

Acknowledgement

We have received useful comments on this report from Tom Dean, Tom Doeppner, Steve Reiss, and Peter Rathmann. 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 applicabilty of these concepts [CCZL:90]. The intial 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 through DARPA contract N39-84-C-211, and, earlier, by IBM Germany. Current support includes 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 Project "Stretch."

References

Since the time is ripe for a movement to megaprogramming, it is not suprising that there is an enormous amount of related work that could be cited. We have only picked up references with which we are immediately familiar, and then are sure to have missed some. Further references can be found in some of the cited references. For object-oriented work references are given in [Wegn:90B]. References to the issues of constructiong large information systems, and the use of agents and mediators in the federated context are found in [WRR+:90]. A bibliography of over 5000 annotated entries in databases, knowledge bases, and related software engineering issues is available for on-line retrieval from Gio@moon.stanford.edu.

[BoSc:90] Barry Boehm and Bill Scherlis: "DARPA Software Engineering Directions"; DARPA-ISTO meeting presentation; June 1990.

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

[Cha:90] Sang K. Cha: "Kaleidoscope: A Cooperative Menu-Guided Query Interface"; *Proc. IEEE Conf. Artifical Intelligence Applications*, Santa Barbara CA, Mar.1990, pp.304–310.

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

[DeMi:89] Linda G. DeMichiel: "Resolving Database Incompatibility: An Approach to Performing Relational Operations over Mismatched Domains"; *IEEE Trans. Knowledge and Data Eng.*, Dec.1989, Vol.1 No.4, pp.485–493.

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

[GuLe:90] R.V. Guha and Douglas Lenat: "CYC, A Midterm report"; Stanford Univ. and MCC, June 1990.

[Hay:85] Patrick Hayes: "Naive Physics 1, The Ontology of Liquids"; in *Formal Theories of the Commonsense World*, Ablex Series in Artificial Intelligence, Eds Hobbs and Moore, 1985.

[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.

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

[LBGSW:88] Barbara Liskov, Toby Bloom, David Gifford, Robert Scheffler, William Weihl: "Communication in the MERCURY System"; MIT LCS Programming Methodology Group 59-1, Revised April 1988.

[LeTr:90] Nancy Leveson, Ed.: *Transactions on Software Engineering*; Special Issue on Formal Methods in Software Engineering, September 1990.

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

[McKi:86] Jock D. MacKinlay: "Automatic Design of Graphical Presentations"; Stanford Un., CSD, PhD Th., TR-CS-86-1138, Dec.1986.

[Mett:90] LtComm. Erik Mettala: "Domain Specific Software Architectures"; DARPA ISTO Principal Investigators meeting , July 1990.

[Mill:56] George A. Miller: "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information"; *The Psychological Review*, Vol.63 No.2, Mar.1956, pp.81–97.

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

[Paul:89] James Paul: "Bugs in the Systems: Problems in Federal Government Software Development and Regulation"; Subcommittee on Investigation and Oversight of the House Committee on Science, Space, and Technology, Government Printing Office, Washington, DC, Sept. 1989.

[Rous:86] Nick Roussopoulos and H. Kang: "Principles and Techniques in the Design of ADMS"; *IEEE Computer*, Vol.19 No.12, Dec.1986 pp.19–25.

[Shoham:88] Yoav Shoham and Nita Goyal: "Temporal Reasoning in Artificial Intelligence"; In: Frontiers of Artificial Intelligene, Morgan-Kaufman, 1988.

[Spec:88] A.Z. Spector, R.F. Pausch,R.F., and G. Bruell: "CAMELOT: A Flexible, Distributed Transaction Processing System"; IEEE Compcon 88, San Francisco, Feb-Mar.1988, pp.432–437.

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

[Wegn:90B] Peter Wegner, "Object-Oriented Megaprogramming"; *ACM Membernet*, Supplement to the CACM, October 1990.

[WRB+:90] Gio Wiederhold, Peter Rathmann, Thierry Barsalou, Byung Suk Lee, and Dallan Quass: "Partitioning and Composing Knowledge"; Information Systems, Vol.15 no.1, 1990, pp.61-72.

[WRR+:90] Wiederhold, Risch, Rathmann, DeMichiel, Chaudhuri, Lee, Law, Barsalou, Quass: "A Mediator Architecture for Abstract Data Access"; Stanford University, Report No.STAN-CS-90-1303, Feb. 1990.

[WWH+:89] Gio CM Wiederhold, Michael G. Walker, Waqar Hasan, Surajit Chaudhuri, Arun Swami, Sang K. Cha, XiaoLei Qian, Marianne Winslett, Linda DeMichiel, and Peter K. Rathmann: "KSYS: An Architecture for Integrating Databases and Knowledge Bases"; in Amar Gupta (ed.) *Integration of Information Systems: Bridging Heterogenous Databases*; IEEE Press, 1989, pp. 72 to 112.

[Zica:90] R. Zicari, L. Tanca, S. Ceri: "Interoperability between a Rule-Based Language and an Object-Oriented Language", preliminary report, to be completed as a Politecnico di Milano Internal Report, P. L. da Vinci, 32, Milano, I20133 Italy, 1990.