CLAM: Composition Language for Autonomous Megamodules

Neal Sample, Dorothea Beringer, Laurence Melloul, Gio Wiederhold

Computer Science Department
Stanford University, Stanford CA 94305
{Nsample, Beringer, Melloul, Gio}@cs.stanford.edu

Abstract. Advances in computer networks that support the invocation of remote services in heterogeneous environments enable new levels of software composition. In order to manage composition at such a high level we envision a need for purely compositional languages. We introduce the CLAM composition language, a megaprogramming language. By breaking up the traditional CALL statement the CLAM language focuses on the asynchronous composition of large-scale, autonomous modules. Furthermore the language has the capability to support various optimizations that are specific to software composition.

1 Introduction

Component software revolutionizes the traditional, programmatic method of building software by introducing a new paradigm in which software is created by leveraging a market of reliable and secure software components [1]. While the component model of software engineering has been promoted for several years [2], it is only in the past few years that infrastructure technology enabling such composition has become stable and sufficiently widespread to encourage serious adoption of the model.

Distributed component software relies on protocols such as DCE [3], CORBA [4], DCOM [1] and Java RMI [4]. Using these protocols, a core software service can be made available to many clients. In order to use these software services, a client program must conform to the Application Program Interface (API) of the desired distribution protocol. However, different servers use different distribution protocols and they do not effectively interoperate. In order to use a distributed software service for composition in the large, a client programmer must be familiar with heterogeneous distributed systems, client-server programming, large-scale software engineering, as well as the application domain. While this is not unrealistic for modest programs composed of small-scale components such as ActiveX controls or JavaBeans [4], as components and assemblies scale upwards, and as more not-technically skilled domain experts would like to make use of distributed components and compose them, this becomes an increasingly unrealistic expectation for most practitioners.

In the CHAIMS (Composing High-level Access Interfaces for Multi-site Software) project, we are attempting to alleviate the knowledge requirements for composition by distinguishing between composition, computation, and distribution. We intend to free the compositional expert from any knowledge of distributed systems and computational programming. At the same time, we recognize that the compositional expert needs more control over the execution and timing of remote method calls as remote server software scales in size. Furthermore, the ability to integrate components that are only accessible by differing distribution protocols is necessary.

With this in mind, we introduce a programming language, CLAM, that is designed for the composition of large-scale modules or services, which we refer to as *megamodules*. A client program that composes various megamodules is known as a *megaprogram*. Megamodules will often be written in different languages, reside on different machines, and be integrated into different distribution systems. Some system is required to bridge the differences in architectures, languages, and distribution systems. We believe that the CPAM (CHAIMS Protocols for Autonomous Megamodules) component of the CHAIMS project adequately addresses those major differences at the runtime level. The support provided by CPAM presents a unique opportunity to investigate a language that strictly addresses composition, without worrying about the aforementioned differences in architectures, module implementation languages, and distribution systems.

We highlight the difference between *coordination* and *composition*. Coordination languages have been studied and used for years, and including languages like SAIL, Ada, and Jovial. Coordination is closely related to *synchronization*, *proper ordering*, and *timing*. Composition is concerned with the act of *combining* part or elements into a whole. Appropriate composition of autonomous services is an important step that must be taken before specific coordination events occur.

2 CLAM in a Sea of Languages

2.1 Objectives

The structure of the CLAM language is motivated by the features a megaprogramming language should support. A language intended for a large-scale environment should implicitly take advantage of parallelism rather than assume sequential execution. We also expect a megaprogramming language to support compile-time as well as run-time optimization. The control structure within the language should reflect the simple elegance achievable when *composition only* is the goal, rather than the traditional composition coupled with computation. Finally, we assume that the environment in which a general megaprogramming language operates is heterogeneous, and therefore the run-time architecture for such a language must bridge competing distribution protocols. Language specific requirements should not impose further limitations on that runtime system.

2.2 Other Approaches

With base-level runtime support for heterogeneous module interoperability provided by a system like CPAM, the focus of the language may turn to module composition. There are myriad languages designed for computation, some of which have support for coordination as well. Such languages and language extensions include FX, Orca, and Opus. These languages were not designed to be *purely* (or seemingly even "primarily") *compositional* languages, however. As such, each of these more "traditional" languages suffers from deficiencies when used for composition. FX requires that all modules are written to conform to the FX language, including data representations. FX codes are also static compositions, and do not have the dynamism expected in an online system [9]. *Static* compositions cannot change or adapt to varying runtime conditions. Orca has its own distinct runtime support for use with its composition primitives, thus radically limits the potential for use of legacy codes in a composed program. The Opus extensions to HPF take a data-centric view that requires a programmer to have intimate knowledge of all data in a process, including typing and semantic meaning, to set up a megaprogram [9, 10].

Developing languages that have some unique features that make them particularly suited to specific domains is not new. LISP was developed to be good at processing lists and has seen widespread application (especially in AI) where that ability is critical. Fortran was designed with significant support for scientific computation and was well suited to that domain, even before additional features such as dynamic allocation of memory were added. CLAM is unique in that its problem domain is *composition*. CLAM does not limit its application potential by focusing on specific implementation schemes. List processing, scientific computing, database access and updating, and all other work on user data is left to the megamodules. CLAM only composes those megamodules and does not restrict itself in ways that any particular megamodule's implementation language necessarily must do.

2.3 MANIFOLD

Like CLAM, the MANIFOLD language is an attempt at a compositional language [11]. MANIFOLD has a control-driven semantic (via "events") which makes it quite different from the data-centric approaches mentioned in 2.2. The MANIFOLD language has the following building blocks: *processes*, *ports*, *streams*, and *events* [12]. Processes are black box views of objects that perform useful work. Ports are access points to processes. Data is passed into and out of ports. Streams are the means to interconnect processes and ports. Events are independent of streams, and convey information through a different mechanism.

MANIFOLD assembles processes using the Ideal Worker Ideal Manager (IWIM) model [12]. In IWIM, compositional codes (known as "managers") select appropriate processes ("workers," in IWIM), even online (at runtime), for a given problem. The processes selected by a manager are workers for that manager. Nesting is possible in the IWIM model, so that one manager's worker may manage workers below it.

IWIM can be viewed like a contractor relationship: a general contractor may hire subcontractors to perform a certain job, and the subcontractors in turn may have additional subcontractors. The general contractor is not concerned with additional layers of contractors/workers, only that the original subcontractor completes the assigned task. MANIFOLD and CLAM both follow this model.

CLAM does not use a black box object view of service providers like MANIFOLD. CLAM treats objects that provide services as entities with exposed methods. These methods can be accessed in a traditional way: invoked with input parameters and with available return values. This can be done in MANIFOLD as well. CLAM, however, extends this simple notion of composition and task atomicity by decomposing the traditional CALL statement, as described in 3.1 and 3.2. In this way, CLAM allows for increased control over scheduling, and allows for runtime inspection of processes. With similar composition features to MANIFOLD, coupled with further CALL statement decomposition, CLAM achieves greater levels of expressiveness. This is especially apparent in 3.2, where we examine optimization opportunities available when the traditional CALL statement is extended with additional functionality.

All MANIFOLD programs can be expressed in CLAM, but the reverse is not true. Because of the special scheduling primitives available in CLAM, more control over process scheduling is possible. Absent these scheduling primitives, and without extensions in CLAM to reuse input parameters to remote invocations, CLAM would be very similar to MANIFOLD.

2.4 A CLAM Language Program

CLAM is not “polluted” with computation and runtime issues because it attempts to be a composition only language. CLAM provides a clean and simple way to achieve the composition painfully achieved within other languages. CLAM has been used to generate working megaprograms compiled with CPAM runtime support. One such megaprogram that frequently appears in the CHAIMS literature finds the least expensive route for transporting goods between two cities [8, 13]. This transportation megaprogram composes five megamodules to achieve this goal.

The following fragment is from an actual transportation megaprogram, for sake of space, some elements have been omitted:

```
// Fragments from megaprogram for finding the cheapest
// route for transporting certain goods between cities

// bind to the megamodules
best_handle         = SETUP ("PickBest")
route_handle        = SETUP ("RouteInfo")
cost_handle         = SETUP ("AirGround")
io_handle           = SETUP ("InputOutput")
best_handle.SETPARAM (criterion = "cost")


// get information from the user about the goods to be
// transported (start and end time, size and weight)
```

```
// and the two desired cities
input_cities_handle = io_handle.INVOKE ("input_cities")
input_goods_handle  = io_handle.INVOKE ("info_goods")
WHILE (input_cities_handle.EXAMINE() != DONE) {}
(mycities = cities) = input_cities_handle.EXTRACT()

// terminate call to "input_cities" within InputOutput
input_cities_handle.TERMINATE()

// get all routes between the two cities
route_handle = route_handle.INVOKE("AllRoutes",
  Pair_of_Cities = cities)

...

//terminate all invocations with "InputOutput" module
io_handle.TERMINATE()
```

3 CLAM Language Semantics

3.1 Parallelism

The compositional programming language CLAM will be used to schedule and organize megamodules. Since remote megamodules can all operate in parallel, the megaprogramming language coordinating these megamodules should be as asynchronous as possible, thus mirroring the everyday world where tasks are often done in parallel. This is particularly important for large-scale megamodules in which each invocation can be expected to be both long running and resource intensive. Synchronous invocation works well for small-scale services. However, as remote services increase in size and power, it is increasingly important that the client has the ability to decide when it wishes to start a specific invocation. In addition, the client must have control returned immediately after starting an invocation so that it can initiate new invocations. When the client needs the results from a particular invocation, it can wait for that invocation, thus synchronizing only when necessary.

In traditional programming languages the primitive to invoke a particular routine, which we refer to as the CALL statement, typically assumes synchrony in execution, forcing possibly parallel tasks into a sequential order or requiring parallelism at the client side. By breaking up the CALL statement into several primitives, we get the asynchrony necessary to support parallel invocation of remote methods out of a sequential client. There is also another reason for breaking up the CALL statement. The CALL statement performs many functions: handling the binding to a remote server, setting local parameters, invoking the desired method, and retrieving the results. For large-scale composition, thinking of the CALL statement as an atomic primitive may make the system unwieldy [5]. Therefore, subdividing the CALL statement into several primitives also gives the megaprogrammer more control over the timing and the execution of the various functions of a CALL statement.

To harness the potential parallelism within a megaprogram, we decompose the traditional CALL statement into the following primitives:

- SETUP

The purpose of the SETUP call is to establish communication with a designated megamodule. SETUP can mean different things depending on the runtime system supporting CLAM. When compiled to CPAM within the CHAIMS system, SETUP primarily establishes communications to megamodules using whatever location scheme is appropriate (e.g. CORBA). Using CLAM with a runtime system like MARS [7], SETUP would likely start a server, rather than simply establish communication to a static one. The SETUP primitive is necessary to direct the runtime client to server/megamodule information at known at compile time (e.g., information taken from the repository we discuss below).

The SETUP calls from the sample CLAM program introduce CLAM handles, e.g.:

```
best_handle  = SETUP ("PickBest")
```

The call to SETUP returns a handle to the megamodule requested, without regard to the locations or implementation of the megamodule. Messy details such as network location and transportation protocols, unimportant to the module composer, are not used in the language, but briefly introduced in 4.1 for the interested reader.

- SETPARAM

The SETPARAM call is used to establish parameters referred to in any method of any particular megamodule that has already been "SETUP." It can also be used to set global variables within a megamodule, i.e. variables used by multiple methods within a single megamodule. If no parameters are set by SETPARAM, the assumption is that the megamodule has suitable default values for an invocation, or the corresponding INVOKE call has all necessary parameters. Many remote procedure calls to major services include various environment variables that are repeated with every invocation. CLAM eliminates that overhead with SETPARAM.

SETPARAM is called using the megamodule handle returned from a SETUP. Again, we see this in the sample program:

```
best_handle.SETPARAM (criterion = "cost")
```

Here, SETPARAM is used to set a global value for `criterion`. Whenever `criterion` is required in any method within a particular megamodule, the value set by SETPARAM may be used for free, or overridden for a particular invocation.

- GETPARAM

The GETPARAM call can return the value, type, and descriptive name of any parameter of any method of a particular megamodule. It can also return any value of that megamodule's global variables. GETPARAM can be done immediately following SETUP to examine initial and/or default values within a megamodule. It can also be done after a method invocation to inspect changes to global variables, if needed. All parameters have at least a global setting.

- INVOKE

The INVOKE call starts the execution of a specified method. Any parameters passed to a method with a call to INVOKE take precedence over parameters already set using SETPARAM. Execution of a specific method returns an "invocation handle." This handle can be used to examine a specific instance of an invocation, and to terminate it as well. We see invocation handles returned in the sample code from 2.4:

```
input_cities_handle = io_handle.INVOKE ("input_cities")
input_goods_handle  = io_handle.INVOKE ("info_goods")

route_handle = route_handle.INVOKE("AllRoutes",
  Pair_of_Cities = cities)
```

With this expressiveness, megaprograms can request multiple instances of the same method from a single megamodule. Single instances can also be terminated with handles. Also, there is no synchronous invocation, so we see the above pairs of modules starting concurrently.

- EXTRACT

The EXTRACT call collects the results of an invocation. A subset of all parameters returned by an invocation can be extracted. Extraction can occur at any point, including partial extractions or extractions of incomplete data. It is up to the megaprogrammer to understand when and if partial extraction is meaningful (we will discuss how another primitive, EXAMINE, reveals that the invocation is DONE or NOT_DONE).

Extraction of data is shown in the transportation code. Note, the return value is not another handle type, but a storage location for the return data:

```
(mycities = cities) = input_cities_handle.EXTRACT()
```

- TERMINATE

The TERMINATE call kills either a running invocation or a connection from a megaprogram to a specific megamodule. Terminating a connection to a megamodule necessarily kills all running invocations within that megamodule initiated by the client issuing the TERMINATE. Invocations belonging to other megaprograms accessing the same megamodule are unaffected, of course. Termination of a specific method from a single client does not invalidate values set by SETPARAM.

Calls of the two possible termination types are shown in 2.4:

```
input_cities_handle.TERMINATE() //acts on an invocation
io_handle.TERMINATE()           //acts on a module
```

The first TERMINATE shown above acts on a single method invocation. The second type of TERMINATE show above ends all invocations within a particular megamodule.

With this breakdown of the method CALL process, CLAM can take advantage of implicit and explicit parallelism. If we assume that all traditional CALL statements are *asynchronous*, some parallelism can be achieved without the decomposition found in CLAM. However, some decomposition is implicit in asynchrony as callbacks are required to retrieve data from invocations. Also, traditional asynchronous CALL statements do not yield the scheduling benefits from the primitives in 3.2. Nor do they allow for SETPARAM type operations.

3.2 Novel Optimization Opportunities

As components increase in size, the ability to schedule and plan for the execution of remote services becomes more critical. Traditional optimization methods have focused, quite successfully, on compile-time optimization. The CLAM language coupled with the CPAM (or another suitable) architecture has the potential to support both run-time and compile-time optimization, thus conforming to the dynamic nature of a distributed environment. Here we extend the work done on dynamic query optimization for databases into software engineering [6].

There are both compile-time and runtime optimizations possible with CLAM. The compile-time optimizations depend of the specific support system, so we focus on runtime optimizations with CLAM. There are four main runtime optimizations we focus on here: simple selection of appropriate megamodules, optimizing setup parameters, scheduling parallel megamodules based on cost functions, and partial extractions.

ESTIMATE for Module Selection. In a widely distributed environment, the availability of megamodules and the allocation of resources they need is beyond the control of the megaprogrammer. Furthermore, several megamodules might exist that offer the same functions. Therefore a client must be able to check the availability of megamodule services and get performance estimates from megamodules prior to the invocation of their services. This needs to be done at run-time, as the compile-time estimation may change by the time the megaprogram is executed. Traditional CALL statements do not consider the notion of execution cost estimates. As such estimates become increasingly important to both module users and providers, explicit language support becomes essential.

The following primitive is added to CLAM to help composers take advantage of runtime selection:

- ESTIMATE

The ESTIMATE call returns an estimation of the would-be cost of a specific method invocation. CLAM recognizes three metrics for estimation: invocation fee, invocation time, and invocation data volume. It is up to the megaprogrammer to use this data in a meaningful way. With the information returned from ESTIMATE, megaprograms can schedule invocations from specific service providers based on costs of their services.

An example of using values from ESTIMATE to steer execution (note, INVEX is short for "invoke and extract," and shown in 3.3):

```
cost1 = method1_handle.ESTIMATE()
cost2 = mathod2_handle.ESTIMATE()
IF (cost1 < cost2) THEN result = method1_handle.INVEX()
ELSE result = method2_handle.INVEX()
```

ESTIMATE provides the CLAM language programmer more scheduling ability than MANIFOLD. This makes ESTIMATE a useful language addition for scheduling, without limiting CLAM in other ways.

By using the ESTIMATE primitive, megaprogram users (or compiled megaprograms) can make online choices about particular megamodules to use. These decisions may be based on factors important during that particular megaprogram invocation. In some instances, importance may lie with the cost of the solution, the time necessary to deliver an acceptable solution, or the data volume returned from a particular query.

Parallel Scheduling. It can be difficult at compile time to appropriately schedule distributed services to optimize performance. Even with compile time estimates of module execution times/costs, actual runtime performance may differ significantly. To combat the problem of compile time naiveté, we include a runtime process examination primitive in CLAM**.**

- EXAMINE

The EXAMINE call is used to determine the state of an invocation. EXAMINE returns an enumerated status to the megaprogram. It can return DONE, PARTIAL, NOT_DONE, and ERROR. The examination of a method before invocation cannot be done, as EXAMINE operates on an invocation handle. Recent discussions about the extension of CLAM indicate that an important addition to EXAMINE would be an indicator of the degree of completion: a value, usable by a megaprogrammer, whose meaning is not enforced by the language. Such an additional return value could be used to indicate the level of progress of an invocation (or anything else of interest), but is not specifically required from megamodules.

EXAMINE is used in the transportation example in 2.4 to synchronize the megaprogram after section of parallel code:

```
WHILE (input_cities_handle.EXAMINE() != DONE) {}
```

EXAMINE may be called with the name of a result value as a parameter, to retrieve information about a specific parameter.

Compile time scheduling can be done in many cases. With EXAMINE, distributed services can be scheduled based on much more accurate runtime information from. Megaprogrammers have language level scheduling ability with CLAM. This is not found in other languages like MANIFOLD.

Progressive Extractions. The EXAMINE primitive allows for online process steering. Speculative scheduling may be done based on estimated times of completion returned from EXAMINE. Advanced scheduling and steering can be achieved as well. Recall that EXAMINE has a second parameter, the meaning of which is defined by the megaprogrammer (and detailed in the repository). A call to EXAMINE may return "PARTIAL" in the first field and "70%" in this second field, indicating a 70% confidence in the result at that point of the invocation. If some level of confidence less than 100% is acceptable for a particular problem, then early extraction based on an EXAMINE statement can save megaprogram users on potentially many fronts, including process time and cost.

Optimizing Setup Parameters. The megaprogrammer may also wish to dynamically check the performance of various megamodules by optimizing various setup parameters, e.g., search parameters or simulation parameters. These parameters may influence the speed and quality of the results, and the megaprogrammer may need to try several settings and retrieve overview results before deciding on the final parameter settings. This is best done during run-time execution. This type of language support is also included in the CLAM specification.

The SETPARAM primitive makes perturbation of input sets easy. By inspecting megamodule progress using EXAMINE (shown in 3.2), users can perform online tests of invocation status. When "tweaking" input parameters to expensive (in terms of fee, time, etc.) processes, partial extraction of data (3.2) and early termination may yield great cost savings. With certain classes of problems, particular setup parameters may not converge on a solution. Using EXAMINE with partial extractions can save users significant costs by *not* computing unacceptable solutions. With many languages, users must wait until a process is complete before testing the viability of the results. There are no primitives for interim process examination.

3.3 Simplicity in Control Flow

CLAM was designed with module composition in mind, and appears to have a very limited set of primitives for megaprogrammers to work with. In reality, the CLAM primitives are a sufficient set tailored specifically to megamodule composition, with little regard for providing functionality beyond composition. Even complex comparisons and altering control flows based on megamodule output require helper modules to analyze and reason about user data.

CLAM strives to remain a language solely for *composition* and not for *computation*. As such, the use of simple CLAM data types for activities other than control flow are quite restricted. Megaprogrammers can assign to a limited set of simple data types (mentioned in 4.2), and make comparisons, but little else. Because of this, the language is very elegant.

Control flow is achieved through the use of simple IF and WHILE constructs. Both are used with boolean expressions composed of comparisons among simple data types. The WHILE loop has the following form:

```
WHILE (Boolean Expression) Statement_list
```

The WHILE is a control mechanism used before making invocations that depend on termination of previous event(s). There are many forms of loops available in different languages, but only the simple WHILE in CLAM. Other loop types, such as DO…WHILE and DO…UNTIL can be constructed from the simple WHILE loop.

The IF statement is a control mechanism used before making invocations which depend on the value of previous result(s) or meta-information like performance data. The IF statement has the following form:

```
IF (Boolean Expression) THEN Statement_list
[ELSE Statement_list]
```

Boolean expressions take many forms. The simplest boolean expression is a single boolean variable. Equality tests between integers, strings, and booleans also yield boolean results (a == b). Also, comparison tests may be done for integers and reals (a<b, a>b, a<=b, a>=b).

By using the IF and WHILE statements, megaprogrammers can write programs in CLAM that have conditional execution paths. These paths may change based on multiple factors including module availability (at setup time), estimations of invocation costs (before potential execution), and progress of particular megamodules (during execution). The sample megaprogram shows the use of a WHILE statement with a megaprogram, and the example for optimization shows the use of an IF statement.

There are two shortcut primitives in CLAM: INVEX (for INVOKE-EXAMINE-EXTRACT) and EXWDONE (for EXTRACT when DONE). These two shortcuts are useful for synchronizing parallel executions, and stalling pipelined megaprograms until critical results become available.

- INVEX

The INVEX call starts the execution of a specified method the same way the INVOKE primitive does. However, INVEX does not return until it collects the results of the invocation. There is currently no possibility to EXAMINE or TERMINATE an INVEX call, though this is still under investigation (there is no reason to EXAMINE or TERMINATE, except in case of error). The INVEX shortcut, operating similarly to the traditional call statement, blocks until method execution is complete and results have been collected. This can be an important short cut when strict synchronicity is desired, or for short methods of helper modules.

- EXWDONE

The EXWDONE call waits until all desired results from a particular invocation are available and then collects those results. There is a clear relationship between EXWDONE and INVEX; INVEX is equivalent to INVOKE immediately followed by EXWDONE. As such, EXWDONE facilitates the addition of instructions between an INVOKE and a corresponding EXTRACT from a single invocation, with the same possibility for synchronization as INVEX.

4 Implementing CLAM

The complete CHAIMS system is based on three main components: CLAM (the language), the CHAIMS compiler, and CPAM (for runtime support). It is a working implementation of the CLAM language. Because a compiler and system supporting the current version of CLAM was developed parallel to CPAM, they coexist naturally. However, the CLAM language is not restricted to use with CPAM and the CHAIMS system. CPAM provides runtime support for myriad communication protocols to CLAM, but is not critical to the language.

As a language, CLAM can compile to alternate runtime systems, such as MARS (Multidisciplinary Application Runtime System), recently developed at the University of Wyoming [7]. MARS, like CPAM, is a self-contained runtime system capable of exchanging data between heterogeneous autonomous megamodules. As such, CLAM is truly an independent part of CHAIMS, with unique language features specifically appropriate for compositional programming.

4.1 CLAM Repository

To keep implementation details away from the language, there must be a clear mapping of names between the megaprogram and information needed by the runtime system. For instance, in the sample megaprogram presented, there are many hidden issues even within the single statement:

```
route_handle = SETUP ("RouteInfo")
```

"`RouteInfo`" is a megamodule that the megaprogrammer wishes to use, but where is it located on the network? What protocols does it communicate with? What are its available invocation methods? The first two issues, location and communication protocols are of no real concern to the megaprogrammer, but there must be a way for the module specified in the CLAM language statements to be located by the runtime system. The third issue, available methods, is more important to the programmer. He must be able to know about methods of megamodules.

To resolve these three issues (locations, protocol, methods), there is a repository where megamodules are registered. The repository provides critical information to the runtime system about location and protocol to the compiler, and provides method information to the megaprogrammer.

The entry in the repository for the megamodule “RouteInfo” shown in the sample megaprogram has the following information:

```
MODULE “RouteInfoMM” CORBA ORBIX sole.stanford.edu
“RouteInfoModule”
```

This repository entry, typical for a CORBA megamodule, consists of the tag “MODULE,” the module name, the tag “CORBA,” the object request broker type (“ORBIX” here), the hostname, and the service. This information is important to the runtime system, but not to the megaprogrammer. The methods available to the megaprogrammer and the parameters used also appear in the repository:

```
METHOD RouteInfoMM.GetRoutes (IN CityPair, RES Routes)
  /* gets all routes between 2 cities */
PARAM RouteInfoMM.CityPair     OPAQUE
  /*data containing a city pair    */
PARAM RouteInfoMM.Routes       OPAQUE
  /*data containing routes         */
```

The method and parameter entries in the repository allow for type checking within the megaprogram. The use of any full-featured standard repository system would be appropriate <<>>>(e.g., from CORBA or X500 realms)<<>>. There is no intrinsic contribution to this project from our particular repository system; it must simply be available, in some form, to the megaprogrammer and to the client program.

4.2 CLAM Data Types

There are six data types in CLAM:

- opaque
- integer
- string
- Boolean
- real
- datetime

The first type, *opaque*, is a complex type. Elements of type opaque are exclusively used as user data (i.e. megamodule data) and are not modifiable with CLAM language statements. Opaque data is transferred between megamodules but never processed in the megaprogram. That data generally cannot be directly examined by a program written in a compositional language without violating the notion of being composition only. Opaque data follows the concepts also found in other compositional languages like MANIFOLD.

The other five types are simple CLAM types. Data of simple type is primarily used for control, and it can be received from megamodules with the primitive EXAMINE (the time parameter of type datetime, data volume of type integer, and fee of type real) and the primitive EXTRACT. EXTRACT allows programs to have special methods in megamodules that take opaque types as input and have simple CLAM types as results. Though CLAM does not provide any arithmetic on complex types, it provides comparisons for simple types.

There are no classes or amalgamated data types in CLAM. This makes sense since CLAM aims to be a purely compositional language. There are no complex calculations or comparisons in the megaprogram, nor are there large data structures within the megaprogram itself that are queried or updated within the megaprogram. Literals may be assigned to the five simple types but, true to its compositional nature, there are no other (non-comparison) operators in CLAM (such as an integer increment operator).

By restricting the data type set in this way, the CLAM language remains elegant. If control decisions are to be made using CLAM data types (perhaps from a call to EXAMINE), they can be done within the megaprogram or by invoking methods of helper modules, e.g. for cost-functions. If decisions must be made about opaque user data, rather than extend CLAM to handle all possible data representations for an arbitrary megamodule, the decision is also pushed to a helper megamodule. This helper module will be able to inspect data, and pass back control information as a CLAM type.

5 Conclusions

The CHAIMS project investigates a new programming paradigm: compositional programming. As increasing numbers of autonomous codes become available to megaprogrammers, the importance of the composition process rises as well. CLAM, developed within the CHAIMS project, is an important piece of the investigation of compositional programming, and represents a purely compositional language.

Does CLAM take advantage of the opportunities presented within a programming paradigm shifting towards composition? We believe it does. Breaking down the traditional CALL statement as CLAM does provides new opportunities for parallelism within megaprograms. Such parallelism, intrinsically important because of the nature of autonomous megamodules, is not achievable with standard languages not designed for composition. CLAM provides a tool to harness the asynchrony necessarily present when dealing with independent information suppliers.

CLAM is simultaneously suited to take advantage of novel optimization opportunities. Simple compile-time optimizations are possible with CLAM, but the possible runtime optimizations will likely prove to be much more significant. As a language, CLAM delivers explicit language support for runtime optimizations. The inclusion of cost estimation methods allows easy online examination of invocation costs. That language-level support means that CLAM can easily handle runtime optimizations not possible in current language approaches based on the traditional CALL routine.

A new composition language should not place increased language restrictions on heterogeneity. Differences in communication protocols should not affect the megaprogrammer. The composition language of choice should not be fundamentally tied to any specific protocol. As mentioned before, the CLAM language does not distinguish between the different protocols at a semantic level. Any need to separate features is done at the level of the compiler or support system. Furthermore, by not committing itself to computations on user data, CLAM is not restricted in its ability to pass data between arbitrary megamodules. The opaque data type in CLAM can handle whatever objects are returned by a megamodule, without restriction.

The compositional programming paradigm represents a next level of abstraction in programming. Machine code can be mapped to assembly language instructions. Assembly can be generated by high level programming languages. High level languages enhance a programmer's ability to generate increasingly complex sets of assembly language code, without adding new assembly language instructions or changing their meaning. High level programming languages are a powerful means to abstract away the tedious details of the assembly language and machine code below them. Composition languages are to high level programming languages what those languages were to assembly and machine code. They enable programmers to use megamodules written in high level languages as a useful construct who's implementation details are not of great concern (like assembly language is to the high level language). CLAM is a step toward this next level of programming.

The CHAIMS project is supported by DARPA order D884 under the ISO EDCS program, with the Rome Laboratories being the managing agent, and also by Siemens Corporate Research, Princeton, NJ. We also thank the referees of COORD'99 for their valuable input, and the various Master's and Ph.D. students that have contributed to the CHAIMS project.

References

1. W. Tracz: "Confessions of a Used Program Salesman"; Addison-Wesley, 1995
2. P. Naur and B. Randell, eds.: "Software Engineering"; Proc. of the NATO Science Committee, Garmisch, Germany, October 7-11, 1968. Proceedings published January 1969
3. W. Rosenberry, D. Kenney and G. Fisher: "Understanding DCE"; O'Reilly, 1994
4. C. Szyperski, "Component Software: Beyond Object-Oriented Programming", Addison-Wesley and ACM-Press New York, 1997
5. G. Wiederhold, P. Wegner, and S. Ceri: "Towards Megaprogramming: A Paradigm for Component-Based Programming"; Communications of the ACM, 1992(11): p.89-99
6. G. Grafe and K. Karen: "Dynamic Query Evaluation Plans"; In James Clifford, Bruce Lindsay, and David Maier, eds., Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, June 1989
7. N. Sample and M. Haines: "MARS: Runtime Support for Coordinated Applications"; Proceedings of the 1999 ACM Symposium on Applied Computing, San Antonio, Texas, Feb-Mar 1999.
8. C. Tornabene, D. Beringer, P. Jain and G. Wiederhold: "Composition on a Higher Level: CHAIMS," *unpublished*, Dept. of Computer Science, Stanford University.
9. H. Bal and M. Haines: "Approaches for Integrating Task and Data Parallelism," Technical Report IR-415, Vrije Universiteit, Amsterdam, December 1996.

10. M. Haines and P. Mehrotra: “Exploiting Parallelism in Multidisciplinary Applications Using Opus,” Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, CA, February 1995.
11. F. Arbab, I. Herman, and P. Spilling: “An Overview of MANIFOLD and its Implementation,” *Concurrency: Practice and Experience*, Volume 5, issue 1, 1993.
12. G. Papadopoulos and F. Arbab: “Modeling Electronic Commerce Activities Using Control Driven Coordination,” Ninth International Workshop on Database And Expert Systems Applications (DEXA 98): Coordination Technologies for Information Systems, Vienna, August 1998.
13. D. Beringer, C. Tornabene, P. Jain, and G. Wiederhold: “A Language and System for Composing Autonomous, Heterogeneous and Distributed Megamodules,” DEXA 98: Large-Scale Software Composition, Vienna, August 1998.