# CPAM, A Protocol for Software Composition

Laurence Melloul, Dorothea Beringer, Neal Sample, Gio Wiederhold
Computer Science Department, Stanford University
{melloul@db, beringer@db, sample@cs, gio@db}.stanford.edu
http://www-db.stanford.edu/CHAIMS

## Abstract

Software composition is critical for building large-scale applications. In this paper, we consider the composition of components that are methods offered by heterogeneous, autonomous and distributed computational software modules made available by external sources. The objective is to compose these methods and build new applications with no integration, hence decreasing the time and the cost needed for producing and maintaining the added functionality. In the following, we describe a high-level protocol which makes it possible to accomplish such software composition. The CPAM protocol can be used on top of various distribution systems. It offers additional features for dealing with the issues of heterogeneity and autonomy, and implements several optimization concepts such as run-time cost estimation of available methods and partial extraction of results.

**Keywords**: Software Composition, Protocol for Composition, Distributed Systems, Autonomous Systems, Optimized Composition

## Introduction

CPAM is a high-level protocol for realizing software composition. CPAM has been defined in the context of the CHAIMS (Compiling High-level Access Interfaces for Multi-site Software) research project [1] at Stanford University in order to build extensive applications by composing large, heterogeneous, autonomous, and distributed software modules.

Software modules are large if they are computation intensive or/and data intensive (computation time ranges from seconds in the case of information requests to days in the case of simulations, and the amount of data transmitted can not be neglected). They are heterogeneous if they are written in different languages (e.g., C++, Java), use different distribution protocols (e.g., CORBA [2], RMI [3]), or run on diverse platforms (e.g., Windows NT, Sun Solaris, HP-UX). Software modules are autonomous if they are developed and maintained independently of one another, and independently of the *composer* who composes them. Finally, they are distributed when they are not located on the same machine server and may be used by more than one client. We will call modules with these characteristics *megamodules* and the methods they offer *services.*

Megamodule composition consists in remotely invoking the services of the composed megamodules in order to produce new services. Software composition differs from software integration in the sense that it preserves megamodules' autonomy. Naturally, the assumption is that megamodule providers are eager to offer their services. This is a reasonable assumption if we consider the business interest that would derive from the use of services, such as a payment of fees or the cut of customer service costs.

Composition of megamodules is becoming crucial for the software Industry. As business competition and software complexity increase, companies have to shorten their software cycle (development, testing, and maintenance) while offering ever more functionality. Because of high software development or integration costs, they are being forced to build large-scale applications by reusing external services (the electronic commerce is a way to access services through the Web) and composing them. These services being offered by independent and geographically distant providers, they are therefore autonomous and most likely heterogeneous and distributed. Also, composition becomes critical when services are large because of cost-effectiveness.

Existing distribution protocols such as CORBA(RMI, DCE [4], DCOM [5]) allow to compose software with different legacy codes but using CORBA(DCE, RMI, DCOM) as the only distribution protocol. The Horus protocol [6] composes heterogeneous protocols in order to add functionality, but at the protocol level only. The ERPs, Enterprise Resource Planning systems (such as SAP R/3, BAAN IV, PeopleSoft), integrate heterogeneous and initially independent systems but do not preserve software autonomy.

CPAM, CHAIMS Protocol for Autonomous Megamodules, has been defined for composing large, heterogeneous, autonomous and distributed software. CPAM deals with the issues of heterogeneity and distribution, preserves the autonomy of megamodules, and allows efficient composition of large-scale services.

# 1. CPAM deals with the issues of Heterogeneity and Distribution

Composition of heterogeneous and distributed software modules implies several constraints. The composition has to support heterogeneous data transmission between megamodules as well as the diverse distribution protocols used by megamodules.

## 1.1 Data heterogeneity

In order for megamodules to exchange data, data needs to be in a common format (a separate research project is exploring ways to map different ontologies [7]). Also, data has to be machine and architecture independent (16 bit architecture versus 32 bit architecture for instance), and transferred between megamodules regardless of the distribution protocol at either end (source or destination). For these reasons, The current version of CPAM requires data to be ASN.1 structures encoded using BER rules [8].

With ASN.1/BER-encoding rules:

1. Simple data types as well as complex data types can be represented as ASN.1 structures,
2. Data can be encoded in a binary format that is interpreted on any machine where ASN.1 libraries are installed,
3. Data can be transported through any distribution system.

It has not been possible to use the CORBA Interface Definition Language or Java classes for instance, to define data types as these definitions respectively require the same CORBA ORB or the RMI distribution protocol at both ends.

In the future, another option for a common data format might be the emerging XML (eXtensible Markup Language) standard [9] which maps data to strings.

## 1.2 Opaque data and visible data

Because ASN.1 data blocks are encoded, we refer to them as BLOBs (Binary Large OBjects). Before being transported, the data is encoded in the source megamodule. It is then sent to the client where it remains as BLOBs. It gets decoded only when it reaches the destination megamodule.

BLOBs being opaque, they are not readable by CPAM nor by the client using CPAM. A client which does composition does not need to interpret the data it receives from a megamodule or sends to a megamodule. Nevertheless, CPAM allows efficient composition especially by giving a client the possibility of making a decision at any point in the program execution (see section 3 below). Such a decision would very likely be based on the data received. The client must therefore be able to read some of the data received. For this purpose, CPAM allows conversions between BLOBs and simple data types. An intermediate megamodule must be used in order to decode a complex data type as the client is not aware of the underlying data type's structure.

### 1.3 Distribution protocol heterogeneity

We saw that encoded ASN.1 data was transferred independently of the distribution protocols used by the megamodules. Nevertheless, a distribution protocol influences connections between clients and servers. CPAM is a high-level protocol which uses existing protocols for establishing connections to megamodules and transporting data between clients and servers. Therefore, CPAM specifications may be implemented on top of various distribution systems. Currently, in the context of CHAIMS we have integrated the following protocols: CORBA, RMI, DCE, *local* C++ and *local* Java (*local* qualifying a server which is not remote).

CPAM assumes that the client is able to talk to the servers through their various distribution protocols on whatever operating systems they are using. The CHAIMS architecture, together with CLAM (CHAIMS Language for Autonomous Megamodules) [10], allows the generation of such a client.

### 1.4 The CHAIMS architecture

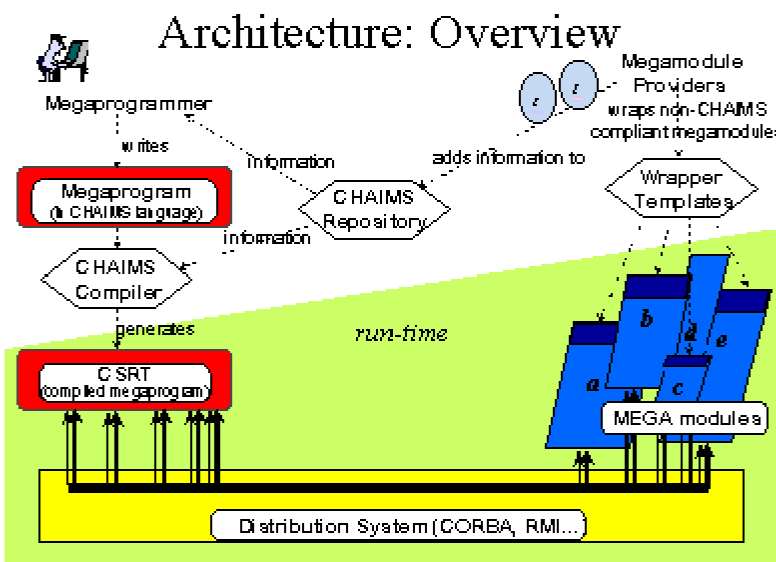


Figure 1.The CHAIMS architecture

Figure 1 describes the CHAIMS architecture. In CHAIMS, the composer is called the *megaprogrammer*, the client program is the *megaprogram* and the compiled program is the *CSRT* (Client Side Run Time).

The green portion corresponds to the run-time system and includes all elements directly related to CPAM:

- The client program which invokes methods offered by the servers,
- The servers (megamodules),
- The distribution system used to transfer data.

The distribution protocol used during a specific communication between the client and a remote server is determined by the distribution protocol used by the server itself. The client has to support such a distribution protocol. CLAM and the CHAIMS compiler [11] produce the necessary client code depending on the information in the CHAIMS repository (see section 2.1).

Both the client and the servers have to follow CPAM specifications. As it is noted in figure 1, megamodules which are not CPAM compliant need to be *wrapped*. The process of wrapping is described in section 4.2.

## 2. CPAM preserves Megamodules' Autonomy

Besides being heterogeneous and distributed, megamodules are autonomous. They are developed and maintained independently from the composer which therefore has no control over them. How can the composer be aware of all services offered by megamodules and of the latest versions of these services without compromising megamodules' autonomy? Also, how do the connections and disconnections between the client and the servers take place? Does the client or the server control the connection?

### 2.1 Information repository

Composition can not be achieved without knowing what services are offered and how to use them. A software user's guide is generally sufficient to know what the purposes of the services are, but does not contain any implementation details about the services. A programmer's guide does contain some implementation details, but these usually represent static information (parameter names and parameter types, for instance). Megamodules, being autonomous and distributed, make it necessary to retain knowledge of dynamic information (e.g., the names of the machines where the services are located).

CPAM requires that the necessary information, both static and dynamic, be gathered into one information repository. Each megamodule provider is responsible for making such a repository available to external users, and for keeping the information up-to-date.

#### 2.1.1 The information repository's content

The Information Repository has to include the following information:

1. Name of server (name of the megamodule), along with its location (machine name) and distribution protocol,
2. Services offered (top-level methods), along with the name and nature (Input or Output) of their parameters,
3. Names of method parameters and global variables, with their explicit types in the case of simple types (a keyword such as *opaque* must be specified in the case of a complex data type).

Such information is necessary for the following reasons:

1. The server machine's location is needed for the client to bind to the server in the case of a remote server (which is likely as megamodules are distributed),
2. The method names as well as the name and nature of the parameters are necessary for making invocations or presetting parameters before invocation,
3. The parameter types are used by the client in case it needs to interpret the data it receives from a megamodule in order to make a decision during the process of composition. CPAM only allows the client to inspect simple types data. The megamodule provider is required to explicitly include simple parameter types in the repository.

### 2.1.2 The scope of parameter names

The scope of parameter names is not only the method where the parameters are used but the whole megamodule. For megamodules offering more than one method, this implies that if two distinct methods have the same parameter name in their lists of parameters, any value preset for this parameter will apply to any use of this parameter in the megamodule. CPAM enlarges the scope of parameter names in order to offer the possibility of presetting all parameters of a megamodule using only one call in the client, hence minimizing data flow (see SETPARAM in section 3.2).

### 2.2 Establishing or terminating a connection with a megamodule

Another issue when composing autonomous megamodules is the ownership of the connection between a client and a server.

Clients are responsible for making a connection to a megamodule and terminating it; servers must be able to handle simultaneous requests from various clients and must be started before such requests arrive. Certain distribution protocols like CORBA include an internal timer which stops a server execution process if no invocations occur after a set time period and instantly starts it when a new invocation arrives.

CPAM defines two primitives in order for a client to establish or terminate a connection to a megamodule. These are **SETUP** and **TERMINATEALL**. SETUP tells the megamodule that a client wants to connect to it; the megamodule generates the necessary internal data structures to handle all future calls from this client. TERMINATEALL notifies the megamodule that the client will no longer use its services; the megamodule kills any ongoing invocations initiated by this client and deletes all related data structures. For both calls, the client is identified with a *clientID*. If for any reason, a client does not terminate a connection to a megamodule, we can assume the megamodule itself will do it after a time-out and a new SETUP will be required from the client before any future invocation.

The information repository, along with the connection rules, ensures that a server's autonomy is preserved and that the clients can use the services offered.

## 3. CPAM allows Efficient Composition of Large-scale Services

CPAM makes it possible to compose services offered by heterogeneous, distributed and autonomous megamodules. Services being large, an even more interesting objective for a client would be to **efficiently** compose these services. CPAM allows efficient composition in the following two ways:

- Invocation sequence optimization
- Data flow minimization between megamodules [12].

### 3.1 Invocation sequence optimization

We have described two primitives of CPAM so far, the two necessary for setting and terminating a connection from a client to a megamodule (SETUP and TERMINATEALL). We have not yet defined the structure of a method invocation nor how the sequence of invocations could be set in the client program.

Because the invocation cost of a large service is a priori high and services are distributed, a random composition of services could be very expensive. The invocation sequence has to be optimized. CPAM has defined its own invocation structure in order to allow parallelism and easy invocation monitoring. Such capabilities add to the possibility of estimating a method cost prior to its invocation and optimize the invocation sequence in the client.

### 3.1.1 Parallelism and invocation monitoring

A traditional procedure call consists in invoking a method and getting its results back in a synchronous way. The calling client waits during the procedure call but the overall structure of the client program remains simple. In contrast, an asynchronous call avoids client waits but makes the client program more complex as it has to be multithreaded. CPAM splits the traditional call statement into four synchronous remote procedure calls which make the overall call behave asynchronously and keeps the client program sequential and simple. The primitives are used for invoking a method (INVOKE), examining an ongoing invocation (EXAMINE), extracting results (EXTRACT), and terminating an invocation (TERMINATE).

These four primitives are described below:

1. **INVOKE** starts the execution of a method, specifying the client's identifier (*clientID*), the name of the method to be invoked, and a set of parameters as a list of name-value pairs. Not every input parameter of the method or global variable used in the method has to be specified in the name-value list. The megamodule takes client-specific values or general default values for missing parameters or global variables (see hierarchical setting of parameters, section 3.2.2). Also, the list of name-value pair is unordered. An INVOKE call returns a *callID* which is used in all subsequent operations on this invocation (EXAMINE, EXTRACT and TERMINATE).

2. The client checks if the results of an INVOKE call are ready using the **EXAMINE** primitive. EXAMINE returns two pieces of information: an *invocation status* which can be any of DONE, NOT_DONE or ERROR, and a description flag whose semantics is megamodule specific. For instance, a flag value could be quantitative and describe the degree of completion of an INVOKE call, or qualitative and specify the degree of resolution a first round of image processing would give.

3. The results of an INVOKE call are retrieved using the **EXTRACT** primitive. A name list is given as input to specify the names of the parameters the client expects. This list is a subset of all possible results of a method. EXTRACT returns a name-value list which contains all the requested results. CPAM does not prevent a client from repeatedly extracting an identical or different subset of results.

4. **TERMINATE** is used to tell a megamodule that the client is no longer interested in a specific invocation. TERMINATE is necessary because the server has no other way to know whether an invocation will be referred to by a client in the future. Indeed, subsequent to an INVOKE, there may be zero, one or more other calls. In case the client is no longer interested in an invocation's results, TERMINATE allows the server to abort an ongoing execution. In case the invocation not only computes results for the client but also generates changes locally, in the server (e.g., in reservation services), it is the responsibility of the megamodule to preserve consistency.

The benefits of having the call statement split into these four primitives are parallelism, simplicity and easy invocation monitoring:

- **Parallelism**: the methods of different megamodules can be executed in parallel, the only restrictions being data flow dependencies. The client program initiates as many invocations as desired and begins collecting results when it needs them.

- **Simplicity**: the client program using CPAM is sequential and simple. It does not have to manage any callbacks from servers (the client is the one which initiates all the calls to the servers, including the ones for getting invocation results).

- Easy invocation monitoring:

    - ♦ **Progress monitoring**: a client can check a method execution progress (EXAMINE), and abort a method execution (TERMINATE). Consider the case where a client has the choice between megamodules offering the same service and arbitrarily chooses one of them for invocation: EXAMINE allows the client to confirm or revoke his choice, perhaps even ending an invocation if another one seems more promising.

    - ♦ **Partial extraction**: a client can extract a subset of the results of a method, only including the elements it needs. CPAM also allows progressive extraction: the client can repeatedly extract new results. Incremental extraction of results is feasible if the megamodule makes a result available as soon as its computation is completed, even before the computation of the next result is done.

    - ♦ **Ongoing processes**: separating method invocation from result extraction as well as from method termination allows ongoing processes, processes which continuously compute or complete results.

#### 3.1.2 Cost estimation

Estimating the cost of a method prior to its invocation augments the probability of making the right invocation at the right time. This is done in CPAM through the **ESTIMATE** primitive. A client asks a megamodule for a cost estimation and then decides whether or not to make the invocation based upon the estimates received. ESTIMATE is very valuable in the case of identical or similar large services offered by more than one megamodule. Indeed, for expensive methods offered by several megamodules, it could be very fruitful to first get an estimate of the invocation cost before choosing one of the methods.

Differences in cost are treated in CPAM as *fees* (amount of money to pay to use a service), *time* (time of a method execution) and *data volume* (amount of data resulting from a method invocation). Since the last two factors are highly run-time dependent, their estimation has to be at run-time as close as possible to the time the method should be invoked. Due to the autonomy of megamodules, the client has no knowledge of or influence over the availability of resources. The ESTIMATE primitive which is offered by the server itself is therefore the only way a client can get most accurate performance and cost information.

The input parameters for ESTIMATE are the *clientID* of the calling client, the name of the method to estimate and a *name list* containing the names of the cost parameters the client is expecting (fee, time or/and data volume). Due to the generic nature of the name list, additional parameters like *location* could be added without changing CPAM. The output parameter of ESTIMATE is a *name-value list* containing the requested estimates.

Parallelism, invocation estimations and invocation examinations are very helpful functionality of CPAM which, when combined, give enough information and flexibility to get an optimized sequence of invocations at run-time. Another optimization factor in CPAM concerns data flow between megamodules.

### 3.2 Data flow minimization between megamodules

Two additional primitives are made available by CPAM in order to minimize data flow: SETPARAM and GETPARAM.

### 3.2.1 Presetting parameters

**SETPARAM** allows setting of method parameters and global variables before a method is invoked. Giving the possibility to separate parameter setting from the invocation itself makes it possible to avoid parameter redundancy during data transfer between megamodules. If a client invokes a method several times with the same parameter values, or invokes several methods which have a common subset of parameter names with the same values, it becomes cost-effective not to transmit the parameter values repeatedly. Let us recall that megamodules are very likely to be data intensive. Also, in the case of methods which have a huge number of parameters, only a few of which are modified at each call (very common in statistical simulations), SETPARAM becomes very useful, perhaps even necessary. SETPARAM's input parameters are the *clientID* and a *name-value list* containing the names and values of the attributes to be set.

Dually, **GETPARAM**, with the input parameters *clientID* and *a name list*, returns client specific settings or default values of parameters and global variables.

### 3.2.2 Hierarchical setting of parameters

CPAM requires that megamodules provide default values for all parameters or global variables they contain, such that a client does not have to specify a value for each of these parameters and global variables with SETPARAM or INVOKE.

CPAM establishes a **hierarchical setting of parameters** within megamodules. A parameter's default value defines the first level of parameter settings (see figure 2). The second level is the client specific setting (SETPARAM). The third level corresponds to the invocation specific setting (parameter value provided for one specific invocation with INVOKE). Invocation specific settings override client specific settings for the time of the invocation, whereas client-specific settings override general default values for the time of the connection. When a method is invoked, the megamodule takes the invocation specific settings for all parameters for which the invocation supplies values; for all other parameters, the megamodule takes the client specific settings if they exist, otherwise it takes the general default values.

**Hierarchical Setting of Parameters**

Example: parameters of method xz

| | start-date | end-date | from | dest | number |
|---|---|---|---|---|---|
| general default values | 1JAN1998 | 1JAN1998 | LAS | BWI | 1 |
| client-specific settings for client *A* | 4OCT1998 | 6OCT1998 | SJO | ZRH | |
| client-specific settings for client *B* | | | | FRA | 2 |
| method *xz*, client *A*: invocation-specific settings | | | | | |
| actual values used in invocation *xz1* | 4OCT1998 | 6OCT1998 | SJO | ZRH | 1 |
| method *xz*, client *A*: invocation-specific settings | | 7OCT1998 | SFO | | |
| actual values used in invocation *xz2* | 4OCT1998 | 7OCT1998 | SFO | ZRH | 1 |
| method *xz*, client *B*: invocation-specific settings | 1DEC1998 | 9DEC1998 | | | |
| actual values used in invocation *xz1* | 1DEC1998 | 9DEC1998 | LAS | FRA | 2 |

Figure 2. Hierarchical setting of parameters

### 3.2.3 Partial extraction of results

EXTRACT is called with an input list of parameter names which is a subset of the list of output parameters of the method which was invoked. This means that only the results needed are extracted. The amount of parameters transferred between megamodules is therefore minimized.

In conclusion, a client does not need to specify all input data or global variables in order to make an invocation, nor does it need to retrieve all available results. This reduces the amount of data transferred between megamodules and optimizes the communication between the client and servers. Megamodules being large and distributed, invocation sequence optimization and data flow minimization are necessary for accomplishing efficient composition.

# 4. How to use CPAM

We have discussed the various primitives of CPAM necessary to efficiently compose megamodules. Two more points have to be mentioned in order to completely define CPAM and the way composition should be done. The first point concerns the client: how should the invocations be ordered in the client program? The second concerns the server: what needs to be done in order to allow a megamodule which is not CPAM compliant to become compliant and support composition?

## 4.1. Invocation ordering constraints

Figure 3 summarizes the nine primitives of CPAM, with their ordering constraints.

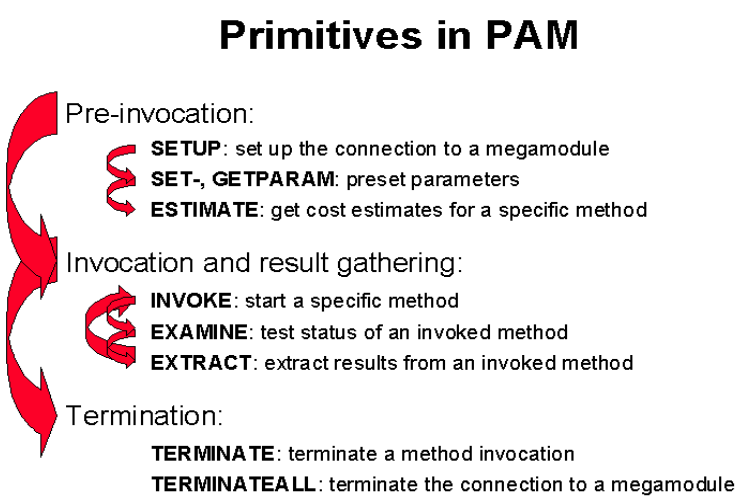


Figure 3. Primitives in CPAM and ordering constraints

CPAM primitives cannot be called in any arbitrary order, but there are only two constraints:

- All primitives apart from SETUP must be preceded by a connection to the megamodule through a call to SETUP which has not yet been terminated by TERMINATEALL,
- The invocation referred to by EXAMINE, EXTRACT, TERMINATE must be preceded by an INVOKE call which has not yet been terminated by TERMINATE.

### 4.2. The CHAIMS wrapper

In case a server does not comply to CPAM specifications, it has to be *wrapped* in order to use the CPAM protocol. Below are summarized CPAM specifications.

#### 4.2.1 CPAM specifications and corresponding functionality

| *CPAM specification* | *Functionality provided* |
| --- | --- |
| 1. Information Repository | Autonomy |
| 2. Underlying distribution protocol | Distribution and Heterogeneity |
| 3. ASN.1/BER data | Heterogeneity |
| 4. SETUP, TERMINATEALL | Connection control |
| 5. Invocation structure (INVOKE, EXAMINE, EXTRACT, TERMINATE) | Parallelism, Invocation monitoring, Partial extraction |
| 6. SETPARAM, Parameter name scoping | Presetting and Hierarchical setting of parameters |
| 7. ESTIMATE | Cost estimation |

If a megamodule does not implement one or more of these specifications, it has to be wrapped. CHAIMS wrapper templates allow a megamodule to become CPAM compliant with a minimum of additional work.

#### 4.2.2 The CHAIMS wrapper

The CHAIMS wrapper is currently implemented as an object (C++ or Java) which serves as a middleman between the network and non-CPAM compliant servers. It implements CPAM specifications in the following way:

- ***Mapping of methods [13, 14] and parameters (point 3 and the INVOKE primitive):*** the wrapper maps methods specified in the information repository to one or more methods of the legacy module. It also maps parameters to ASN.1 data structures, preserving default values assigned in the legacy modules (or adding them if they were not assigned). Both mapping are manually coded. ASN.1 BER-encoding/decoding is done automatically through ASN.1 libraries.

- ***Generation of internal structures to handle client invocations and connections (points 4, 5, 6):*** each call to SETUP generates the structures necessary to store client and invocation related information in the wrapper. Such information eventually includes parameters' and global variables' default values, clientIDs and any client-specific preset values, callIDs, invocations' statuses, and invocations' results. Since invocation related information is stored until a TERMINATE call occurs, repetitive invocation examinations and extractions are possible. The generated structures are deleted only when a call to TERMINATEALL occurs.

- ***Implementation of the ESTIMATE primitive for cost estimation (point 7):*** for each method whose cost estimation is not provided by the server, the ESTIMATE primitive returns an average of the costs of the previous calls of that method, by default.

- ***Threading of invocations:*** to ensure parallelism and respect asynchrony in the legacy code, the CHAIMS wrapper spawns a new thread for each invocation.

Legacy code should include pertinent information for the ESTIMATE and EXAMINE primitives in order for a client to completely take benefit of CPAM optimization functionality through consistent estimation and control.

## Conclusion

CPAM is a high-level protocol for composing megamodules. It deals with heterogeneity and distribution mainly by transferring data as encoded ASN.1 structures. It preserves megamodules' autonomy by collecting services information in an information repository. Most important, CPAM allows efficient composition of large-scale services by optimizing the invocation sequence and minimizing data flow between megamodules. As CPAM is focused on composition, it does not provide support for type checking, recovery or security. These services could be obtained by orthogonal systems or by integrating CPAM into a larger protocol.

A successful utilization of CPAM for realizing composition is the Transportation example implemented within the CHAIMS system. The example consists in finding the best way for transporting goods between two cities. The composer uses services from four heterogeneous and autonomous megamodules. The client program is written in CLAM. It is generated through the CHAIMS compiler and uses CPAM. A second version of the Transportation example is under implementation and will include optimization functionality as cost estimation, partial extraction and qualitative and quantitative invocation examination.

Even more optimization could be achieved by automated scheduling of composed services which use the CPAM protocol. Automation, while not disabling optimizations that are based on domain expertise, will discharge the composer from lower level scheduling tasks. In a large-scale and distributed environment, resources are likely to be relocated, and their available capacity depends on aggregate usage. Invocation scheduling and data flow optimization needs to take into account such constraints. The CPAM protocol gives the necessary information for allowing automated scheduling of composed software at run-time.

## References

[1] G. Wiederhold, P. Wegner and S. Ceri: "Towards Megaprogramming: A Paradigm for Component-Based Programming"; Communications of the ACM, 1992(11): p89-99

[2] J. Siegel: "CORBA fundamentals and programming"; Wiley New York, 1996

[3] C. Szyperski: "Component Software: Beyond Object-Oriented Programming"; Addison-Wesley and ACM-Press New York, 1997

[4] W. Rosenberry, D. Kenney and G. Fisher: "Understanding DCE"; OReilly, 1994

[5] D. Platt: "The Essence of COM and ActiveX"; Prentice-Hall, 1997

[6] R. Van Renesse and K. Birman: "Protocol Composition in Horus"; TR95-1505, 1995

[7] J. Jannink, S. Pichai, D. Verheijen and G. Wiederhold: "Encapsulation and Composition of Ontologies"; submitted

[8] "Information Processing -- Open Systems Interconnection -- Specification of Abstract Syntax Notation One" and "Specification of Basic Encoding Rules for Abstract Syntax Notation One", International Organization for Standardization and International Electrotechnical Committee,

International Standards 8824 and 8825, 1987

[9] "Extensible Markup Language (XML), 1.0", Recommendation of the World Wide Web Consortium, February 1998

[10] N. Sample, D. Beringer, L. Melloul and G. Wiederhold: "The coordination language CLAM"; submitted

[11] L. Perrochon, G. Wiederhold and R. Burback: "A compiler for Composition: CHAIMS"; Fifth International Symposium on Assessment of Software Tools and Technologies (SAST '97), Pittsburgh, June 3-5, 1997

[12] D. Beringer, C. Tornabene, P. Jain and G. Wiederhold: "A Language and System for Composing Autonomous, Heterogeneous and Distributed Megamodules"; DEXA International Workshop on Large-Scale Software Composition, August 28, 1998, Vienna Austria

[13] Birell, A.D. and B.J. Nelso: "Implementing Remote Procedure Calls"; ACM Transactions on Computer Systems, 1984. 2(1): p. 39-59

[14] ISO, "ISO Remote Procedure Call Specification", ISO/IEC CD 11578 N6561, 1991