CPAM, A Protocol for Software Composition

Laurence Melloul, Dorothea Beringer, Neal Sample, Gio Wiederhold

Computer Science Department, Stanford University
Gates Computer Science building 4A, Stanford, CA 94305, USA
{melloul, beringer, nsample, 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 while preserving the autonomy of the software modules. This would decrease the time and cost needed for producing and maintaining the added functionality. In the following, we describe a high-level protocol that enables software composition. CPAM, CHAIMS Protocol for Autonomous Megamodules, may be used on top of various distribution systems. It offers additional features for supporting module heterogeneity and preserving module autonomy, and also implements several optimization concepts such as cost estimation of methods and partial extraction of results.

1 Introduction

CPAM, the CHAIMS Protocol for Autonomous Megamodules, 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) [1] research project 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 (computation time may range from seconds in the case of information requests, to days in the case of simulations) or/and data intensive (the amount of data can not be neglected during transmissions). They are heterogeneous if they are written in different languages (e.g., C++, Java), use different distribution protocols (e.g., CORBA [2], RMI [3], DCE [4], DCOM [5]), or run on diverse platforms (e.g., Windows NT, Sun Solaris, HP-UX). Modules are autonomous if they are developed and maintained independently of one another, and independently of the *composer* who composes them. Finally, software modules are distributed when they are not located on the same server and may be used by more than one client. We will call modules with these characteristics *megamodules,* and the methods they offer *services.*

In this paper, we focus on the composition of megamodules for two reasons:

- service providers being independent and geographically distant, software modules are autonomous and most likely heterogeneous and distributed,
- because of cost-effectiveness, composition is critical when services are large.

M. Jarke, A. Oberweis (Eds.): CAiSE'99, LNCS 1626, pp. 11-25, 1999.
© Springer-Verlag Berlin Heidelberg 1999

Megamodule composition consists of remotely invoking the services of the composed megamodules in order to produce new services. Composition differs from integration in the sense that it preserves megamodule 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 and *composing* them. Global information systems such as the Web and global business environments such as electronic commerce foreshadow a software development environment where developers would access and reuse services offered on the Web, combine them, and produce new services which, in turn, would be accessed through the Web.

Existing distribution protocols such as CORBA, RMI, DCE, or DCOM allow users to compose software with different legacy codes but using CORBA, RMI, DCE, or DCOM as just the distribution protocol underneath. The Horus protocol [6] composes heterogeneous protocols in order to add functionality at the protocol level only. The ERPs, Enterprise Resource Planning systems, such as SAP R/3, BAAN IV, and PeopleSoft, integrate heterogeneous and initially independent systems but do not preserve software autonomy. None of these systems simultaneously supports heterogeneity and preserves software autonomy during the process of composition in a distributed environment.

CPAM has been defined for accomplishing megamodule composition. In the following, we describe how CPAM supports megamodule heterogeneity (section 2), how it preserves megamodule autonomy (section 3), and how it enables optimized composition of large-scale services (section 4). We finally explain how to use CPAM, and provide an illustration of a client doing composition in compliance with the CPAM protocol (section 5).

2 CPAM Supports Megamodule Heterogeneity

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

2.1 Data Heterogeneity

In order for megamodules to exchange information, data need 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 other definition languages such as CORBA Interface Definition Language or Java classes to define data types because these definitions respectively require that the same CORBA ORB or the RMI distribution protocol be supported at both ends of the transmission.

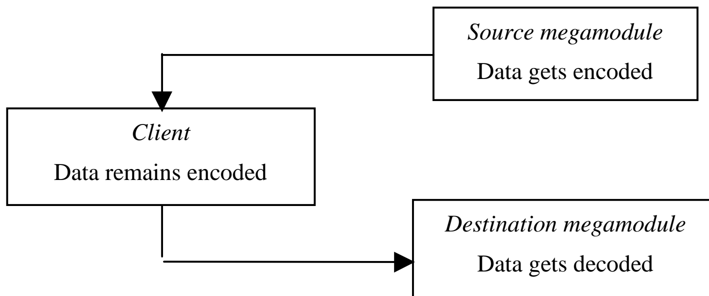

Fig. 1. Data transfer, Opaque data

2.2 Opaque Data

Because ASN.1 data blocks are encoded in binary format, we refer to them as BLOBs (Binary Large OBjects). BLOBs being opaque, they are not readable by CPAM. A client doing composition only (the client gets and transmits the data, with no computation in between) does not need to interpret the data it receives from a megamodule, or sends to another megamodule. Therefore, as shown in Fig. 1, before being transported, data is encoded in the source megamodule; it is then sent to the client where it remains a BLOB, and gets decoded only when it reaches the destination megamodule.

A client that would have the knowledge of the megamodule definition language could convert the blobs into their corresponding data types, and read them. It would then become its responsibility to encode the data before sending it to another megamodule.

2.3 Distribution Protocol Heterogeneity

Both data transportation and client-server bindings are dependent on the distribution system used. CPAM is a high-level protocol that is implemented on top of existing

distribution protocols. Since its specifications may be implemented on top of more than one distribution protocol within the composed application, CPAM has to support data transmissions and client-server connections across different distribution systems.

We mentioned that encoded ASN.1 data could be transferred between the client and the megamodules independently of the distribution protocols used at both ends. Regarding client-server connections, CPAM assumes that the client is able to simultaneously support the various distribution systems of the servers it wishes to talk to. The CHAIMS architecture, along with the CHAIMS compiler [9], enables the generation of such a client. This process is described in next section. Currently, in the context of CHAIMS, a client can simultaneously support the following protocols: CORBA, RMI, *local* C++ and local Java (*local* qualifying a server which is not remote).

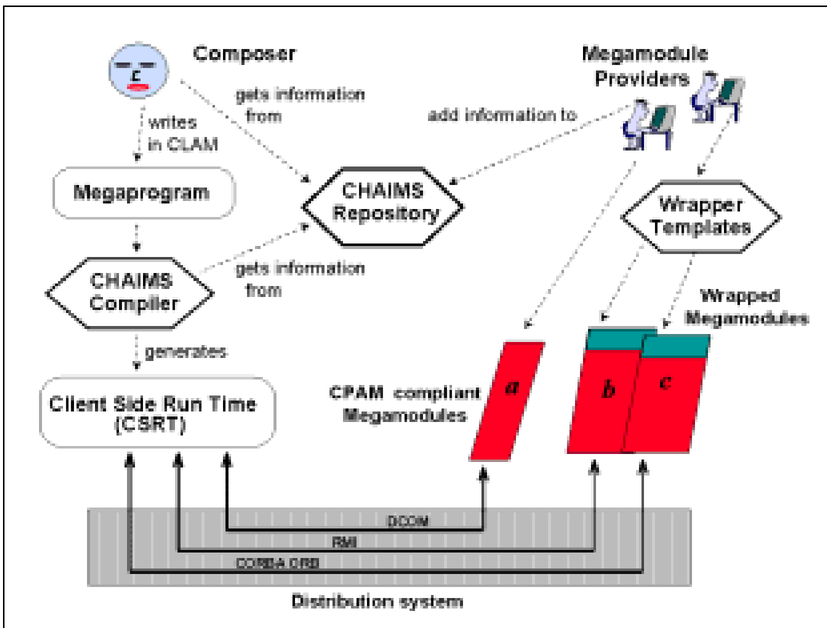

Fig. 2. The CHAIMS architecture

2.4 The CHAIMS Architecture

Figure 2 describes the CHAIMS architecture. In CHAIMS, the client program is the *megaprogram* and the compiled client program is the *CSRT* (Client Side Run Time). Also, server information repositories are currently merged into one unique CHAIMS repository.

The distribution protocol used during a specific communication between the client and a remote server is the one of the server itself, and must be supported by the client. In the context of CHAIMS, the composer writes a simple megaprogram in CLAM (CHAIMS Language for Autonomous Megamodules) [10], a composition only

language. This program contains the sequence of invocations to the megamodules the composer wishes to compose (an example of megaprogram is given in section 5.3). The CHAIMS compiler parses the megaprogram and generates the whole client code necessary to simultaneously bind to the various servers (CSRT). Server specifications such as the required distribution protocol are contained in the CHAIMS repository and are accessible to the CHAIMS compiler.

Both the client and the servers have to follow CPAM specifications. As it is noted in Fig. 2, megamodules that are not CPAM compliant need to be *wrapped*. The process of wrapping is described in section 5.2.

3 CPAM Preserves Megamodule Autonomy

Besides being heterogeneous and distributed, megamodules are autonomous. They are developed and maintained independently from the composer who 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 megamodule autonomy? Also, how do the connections between the client and the server take place? Which of the client or the server controls the connection? After specifying these two points, we will briefly described several consistency rules that will ensure offered services are not updated by the server without the client being aware of it.

3.1 Information Repository

Composition can not be achieved without knowing what services are offered and how to use them. The composer could refer to an application user's guide to know what the purposes of the services available are. He/she could also refer to the application programmer's guide to get the implementation details about the services. Nonetheless, the composer would only get static information such as service description and method input/output parameter names and types. Megamodules being autonomous and distributed, make it compulsory to also retain dynamic information about the services, such as the name of the machines where the services to be composed are located.

CPAM requires that the necessary megamodule 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. It is also the megamodule provider's responsibility to actually offer the services and the quality it advertises.

Information Repository Content. The information repository has to include the following information:

1. Logical name of the service (i.e., megamodule), along with the machine location and the distribution protocol used, in order for the client to bind to the server,

2. Names of the services offered (top-level methods), along with the names and nature (input or output) of their parameters, in order for the client to make invocations or preset parameters before invocation.

Scope of Parameter Names. The scope of parameter names is not restricted to the method where the parameters are used, but rather to the whole megamodule. For megamodules offering more than one method, this implies that if two distinct methods have the same parameter name in their list 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 one call only in the client, hence minimizing data flow (see section 4.2).

3.2 Establishing and Terminating a Connection with a Megamodule

Another issue when composing autonomous megamodules is the ownership of the connection between a client and a server. Autonomous servers do not know when a client wishes to initiate or terminate a connection. In CPAM, clients are responsible for making a connection to a megamodule and terminating it. Nonetheless, 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 that 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; TERMINATEALL notifies the megamodule that the client will no longer use its services (the megamodule kills any ongoing invocations initiated by this client). 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.

3.3 Consistency

Megamodules being autonomous, they can update services without clients or composers being aware of the modifications brought to the services. The best way a client becomes aware of updates in the server is still under investigation (one option could be to have such changes mentioned in the repository). Nevertheless, it should not be the responsibility of the megamodule provider to directly notify clients and composers of service changes since we do want to preserve megamodule autonomy.

Once the composer knows what modifications were brought to the services, he/she can accordingly upgrade the client program. In CHAIMS, the composer upgrades the megaprogram, which the CHAIMS compiler recompiles in order to generate the updated client program.

It is the responsibility of the service provider not to update the server while there are still clients connected to it. The server must first ask clients to disconnect or wait for their disconnection before upgrading megamodules.

The information repository and the connection and consistency rules ensure that server autonomy is preserved and that clients are able to use offered services.

4 CPAM Enables 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 enables efficient composition in the following two ways:

- Invocation sequence optimization
- Data flow minimization between megamodules [11].

4.1 Invocation Sequence Optimization

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, added to the possibility of estimating a method cost prior to its invocation, enable optimization of the invocation sequence in the client.

Invocation Structure in CPAM. A traditional procedure call consists of invoking a method and getting its results back in a synchronous way: the calling client waits during the procedure call, and 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 has to be multithreaded. CPAM splits the traditional call statement into four synchronous remote procedure calls that make the overall call behave asynchronously while keeping the client program sequential and simple. These procedure calls have also enabled the implementation of interesting optimization concepts in CPAM, such as partial extraction and progress monitoring.

The four procedure calls are INVOKE, EXAMINE, EXTRACT, and TERMINATE:

1. *INVOKE* starts the execution of a method applied to a set of input parameters. Not every input parameter of the method has to be specified as the megamodule takes client-specific values or general hard-coded default values for missing parameters (see hierarchical setting of parameters, section 4.2). An INVOKE call returns an invocation identifier, 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* and an *invocation progress*. The invocation status can be any of DONE, NOT_DONE, PARTIAL, or ERROR. If it is either PARTIAL or DONE, then respectively part or all of the results of the invocation are ready and can be extracted by the client. Invocation progress' semantics is megamodule specific. For instance, progress information could be quantitative and describe the degree of completion of an

INVOKE call, or qualitative (e.g., 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. Only the parameters specified as input are extracted, and only when the client wishes to extract results, can it do so. CPAM does not prevent a client from repeatedly extracting an identical or a 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 the client in the future. In case the client is no longer interested in an invocation's results, TERMINATE makes it possible for the server to abort an ongoing execution. In case the invocation has generated persistent changes in the server, it is the responsibility of the megamodule to preserve consistency.

Parallelism and Invocation Monitoring. The benefits of having the call statement split into the four primitives mentioned above are parallelism, simplicity, and easy invocation monitoring:

- *Parallelism*: thanks to the separation between INVOKE and EXTRACT in the procedure call, the methods of different megamodules can be executed in parallel, even if they are synchronous, the only restrictions being data flow dependencies. The client program initiates as many invocations as desired and begins collecting results when it needs them. Figure 3 illustrates the parallelism that can be induced on synchronous calls using CPAM. Similar parallelism could also be obtained with asynchronous methods.

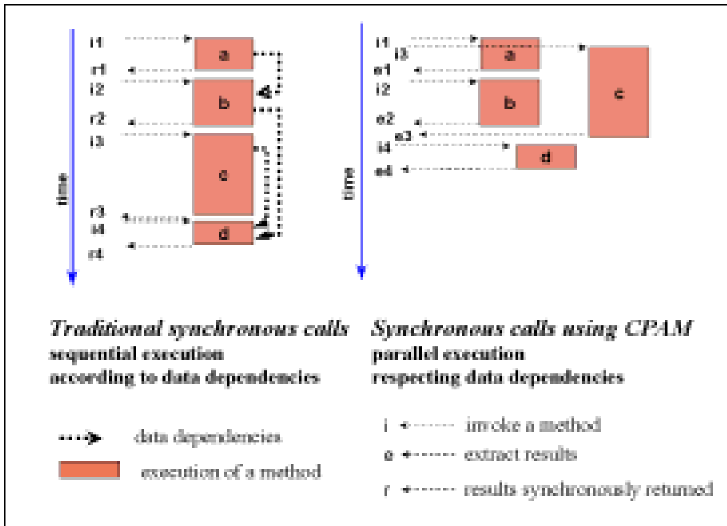

Fig. 3. Split of the procedure call in CPAM and parallelism on synchronous calls

- *Simplicity*: the client program using CPAM consists of sequential invocations of CPAM primitives, and is simple. It does not have to manage any callbacks of

asynchronous calls from the 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 its 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. CPAM also allows progressive extraction: the client can incrementally extract results. This is feasible if the megamodule makes a result available as soon as its computation is completed (and before the computation of the next result is), or becomes significantly "more accurate." Incremental extraction could also be used for terminating an invocation as soon as its results are satisfying, or conversely for verifying the adequacy of large method invocations and maybe terminating them as soon as results are not satisfying.
  - *Ongoing processes:* separating method invocation from result extraction and method termination enables clients to monitor ongoing processes (processes that continuously compute or complete results, such as weather services).

With very few primitives to learn, the composer can write simple client programs, still benefiting from parallelism and easy invocation monitoring. CPAM offers one more functionality in order to optimize the invocation sequence in the client program: invocation cost estimation.

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 enabled in CPAM through the ESTIMATE primitive. Due to the autonomy of megamodules, the client has no knowledge of or influence over the availability of resources. The ESTIMATE primitive, which is provided by the server itself, is the only way a client can get the most accurate method performance and cost information.

A client asks a megamodule for a method cost estimation and then decides whether or not to make the invocation based upon the estimate 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. Of course, there is no guarantee on the estimate received (we can assume that a service invocation which is not in concordance with the estimate previously provided to a client will not be reused by the client).

Cost estimates are treated in CPAM as *fees* (amount of money to pay to use a service, in electronic commerce for instance), *time* (time of a method execution) and/or *data volume* (amount of data resulting from a method invocation). Since the last two factors are highly run-time dependent, their estimation should be at run-time, as close as possible to the time the method could be invoked. Other application-specific parameters like *server location*, *quality of service*, and *accuracy of estimate* could be added to the output estimate list in the server (and in the information repository), without changing CPAM specifications.

Parallelism, invocation estimates and invocation examinations are very helpful functions of CPAM which, when combined, give enough information and flexibility to get an optimized sequence of invocations at run-time. Megamodule code should return pertinent information with the ESTIMATE and EXAMINE primitives in order for a client to completely benefit from CPAM through consistent estimation and control.

Another factor for optimized composition concerns data flow between megamodules.

4.2 Data Flow Minimization between Megamodules

Partial extraction enables clients to reduce the amount of data returned by an invocation. CPAM also makes it possible to avoid parameter redundancy when calling INVOKE thanks to parameter presetting and hierarchical setting of parameters.

Presetting Parameters. CPAM's *SETPARAM* primitive sets method parameters and global variables before a method is invoked. For a client which invokes a method with the same parameter value several times consecutively, or invokes several methods which have a common subset of parameter names with the same values, it becomes cost-effective to not transmit the same parameter values repeatedly. Let us recall that megamodules are very likely to be data intensive. Also, in the case of methods which have a very large number of parameters, only a few of which are modified at each call (very common in statistical simulations), the SETPARAM primitive becomes very advantageous. Finally, presetting parameters is useful for setting a specific context before estimating the cost of a method.

GETPARAM, the primitive dual, returns client specific settings or default values of the parameters and global variable names specified in its input parameter list.

Hierarchical Setting of Parameters. CPAM establishes a *hierarchical setting of parameters* within megamodules (see Fig. 4). A parameter's default value (most likely hard-coded) defines the first level of parameter settings within the megamodule. The second level is the client specific setting (set by SETPARAM). The third level corresponds to the invocation specific setting (parameter value provided through one specific invocation, by INVOKE). Invocation specific settings override client specific settings for the time of the invocation, and 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, and the megamodule general default values otherwise. For this reason, CPAM requires that megamodules provide default values for all parameters or global variables they contain.

Example: parameters of method "reservation"

	start-date	end-date	from	to	seats
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
invocation-specific settings for client A					
actual values used during invocation	4OCT1998	6OCT1998	SJO	ZRH	1
invocation-specific settings for client A		7OCT1998	SFO		
actual values used during invocation	4OCT1998	7OCT1998	SFO	ZRH	1
invocation-specific settings for client B	1DEC1998	9DEC1998			
actual values used during invocation	1DEC1998	9DEC1998	LAS	FRA	2

Fig. 4. Hierarchical setting of parameters

In conclusion, a client does not need to specify all input data or global variables used in a method in order to invoke that method, nor does it need to repeatedly transmit the same data for all method invocations which use the same parameters' values. Also, a client need not retrieve all available results. This reduces the amount of data transferred between megamodules.

Megamodules being large and distributed, invocation sequence optimization and data flow minimization are necessary for efficient composition.

5 How to Use CPAM

We have so far discussed the various CPAM primitives necessary to do composition. Still, we have not yet described the primitives' syntax nor the constraints a composer would have to follow in order to write a correct client program. Another point would concern the service provider: what does he/she need to do in order to convert a module which is not CPAM compliant to a CPAM compliant megamodule that supports CPAM specifications?

CPAM primitives' syntax is fully described and can be found under the CHAIMS Web site at http://www-db.stanford.edu/CHAIMS. In this section, we describe the primitive ordering constraints, the CHAIMS wrapper templates, and provide a client program example which complies with CPAM, and is written in CLAM.

5.1 Primitives Ordering Constraints

CPAM primitives cannot be called in any arbitrary order, but they only have to follow 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, or TERMINATE must be preceded by an INVOKE call (which has not yet been terminated by TERMINATE).

Figure 5 summarizes CPAM primitives and the ordering relations.

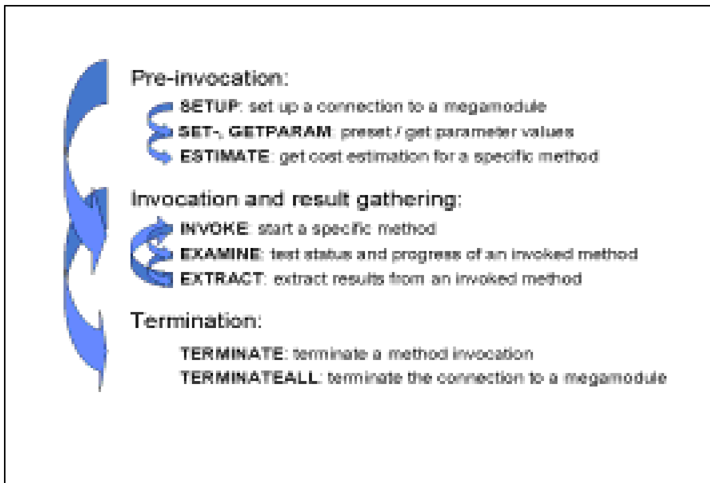

Fig. 5. Primitives in CPAM and invocation ordering

5.2 The CHAIMS Wrapper

In case a server does not comply with CPAM specifications, it has to be *wrapped* in order to use the CPAM protocol. The CHAIMS wrapper templates allow a megamodule to become CPAM compliant with a minimum of additional work.

The CHAIMS wrapper templates are currently implemented as a C++ or Java object which serves as a middleman between the network and non-CPAM compliant servers. They implement CPAM specifications in the following way:

1. *Mapping of methods [12] [13] and parameters:* 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).
2. *Threading of invocations:* to ensure parallelism and respect asynchrony in the legacy code, the CHAIMS wrapper spawns a new thread for each invocation.

3. *Generation of internal data structures to handle client invocations and connections, and support hierarchical setting of parameters:* each call to SETUP generates the necessary data structures to store client and invocation related information in the wrapper. Such information includes client-specific preset values, and the status, progress and results for each invocation. The generated data structures are deleted when a call to TERMINATEALL occurs.
4. *Implementation of the ESTIMATE primitive for cost estimation:* for each method whose cost estimation is not provided by the server, the ESTIMATE primitive by default returns an average of the costs of the previous calls of that method. It could also include dynamic information about the server and the network, such as machine server load, network traffic, etc.
5. *Implementation of the EXAMINE primitive for invocation monitoring:* by default, only the status field is returned. For the progress information to be set in the wrapper, the server has to give significant information.
6. *Implementation of all other CPAM primitives:* SETUP, GET/SETPARAM, INVOKE, EXTRACT, TERMINATE, and TERMINATEALL.

The current CHAIMS wrapper templates automatically generate the code to ensure points 2 to 6. Only requirement 1 needs manual coding (except for BER-encoding/decoding, which is automatically done by ASN.1 libraries).

5.3 Example of a Client Using CPAM

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 five heterogeneous and autonomous megamodules. The client program is written in CLAM, and the CSRT generated through the CHAIMS compiler is in compliance with CPAM. A second example is under implementation. It computes the best design model for an aircraft system, and includes optimization functionality as cost estimation, incremental result extraction and invocation progress examination.

Below is given a simplified version of the Transportation megaprogram (Fig. 6). Heterogeneity and distribution characteristics of the composed megamodules are specified as follows: locality (Remote or Local), language, and protocol.

6 Conclusion

CPAM is a high-level protocol for composing megamodules. It supports heterogeneity especially by transferring data as encoded ASN.1 structures, and preserves megamodule autonomy by collecting service information from an information repository, and by subsequently using the generic invocation primitive of CPAM in order to INVOKE services.

Most importantly, CPAM enables efficient composition of large-scale services by optimizing the invocation sequence (parallelism, invocation monitoring, cost estimation), and minimizing data flow between megamodules (presetting of parameters, hierarchical setting of parameters, partial extraction). As CPAM is currently focused on composition, it does not provide support for recovery or security.

These services could be obtained by orthogonal systems or by integrating CPAM into a larger protocol.

```
Transportationdemo
BEGINCHAIMS
io   = SETUP ("io")           // Remote, Java, CORBA ORBACUS
math = SETUP ("MathMM")       // Local, Java
am   = SETUP ("AirMM")        // Remote, C++, CORBA ORBIX
gm   = SETUP ("GroundMM")     // Remote, C++, CORBA ORBIX
ri   = SETUP ("RouteInfoMM")  // Remote, C++, CORBA ORBIX

// Get type and default value of the city pair parameter
(cp_var = CityPair) = ri.GETPARAM()

// Ask the user to confirm/modify the source and destination cities
ioask = io.INVOKE ("ask", label = "which cities", data = cp_var)
WHILE ( ioask.EXAMINE() != DONE )  {}
(cities_var = Cities) = ioask.EXTRACT()

// Compute costs of the route by air, and by ground, in parallel
acost = am.INVOKE ("GetAirTravelCost", CityPair = cities_var)
gcost = gm.INVOKE ("GetGdTravelCost", CityPair = cities_var)

// Make other invocations (e.g., Check weather)
….

// Extract the two cost results
WHILE ( acost.EXAMINE() != DONE )  {}
(ac_var = Cost) = acost.EXTRACT()
WHILE ( gcost.EXAMINE() != DONE )  {}
(gc_var = Cost) = gcost.EXTRACT()

// Compare the two costs
lt = math.INVOKE ("LessThan", value1 = ac_var, value2 = gc_var)
WHILE ( lt.EXAMINE() != DONE )  {}
(lt_bool = Result) = lt_ih.EXTRACT()

// Display the smallest cost
IF ( lt_bool == TRUE ) THEN
{ iowrite = io.INVOKE ("write", data = ac_var) }
ELSE
{ iowrite = io.INVOKE ("write", data = gc_var)  }

am.TERMINATE()
ri.TERMINATE()
gm.TERMINATE()
io.TERMINATE()
math.TERMINATE()
ENDCHAIMS
```

Fig. 6. CLAM megaprogram to calculate the best route between two cities

In the future, we plan to enable even more optimization through automated scheduling of composed services that use the CPAM protocol within the CHAIMS system. Automation, while not disabling optimizations that are based on domain expertise, will discharge the composer from parallelism or 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 need to take into account such constraints. The CPAM protocol can give sufficient information to the compiler or the client program for enabling automated scheduling of composed software at compile-time, and more significantly, 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. 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
10. N. Sample, D. Beringer, L. Melloul and G. Wiederhold: “CLAM: Composition Language for Autonomous Megamodules”; Third Int'l Conference on Coordination Models and Languages, COORD'99, Amsterdam, April 26-28, 1999
11. 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
12. Birell, A.D. and B.J. Nelso: "Implementing Remote Procedure Calls"; ACM Transactions on Computer Systems, 1984. 2(1): p. 39-59
13. ISO, "ISO Remote Procedure Call Specification", ISO/IEC CD 11578 N6561, 1991