A Language and System for Composing Autonomous, Heterogeneous and Distributed Megamodules

Dorothea Beringer, Catherine Tornabene, Pankaj Jain, Gio Wiederhold

Stanford University, Computer Science Departement, Stanford, CA 94306, USA
{beringer, catherin, pjain, gio}@db.stanford.edu

Abstract

New levels of software composition become possible through advances in distributed communication services. In this paper we focus on the composition of megamodules, which are large distributed components or computation servers that are autonomously operated and maintained. The composition of megamodules offers various challenges. Megamodules are not necessarily all accessible by the same distribution protocol (such as CORBA, DCOM, RMI and DCE). Their concurrent nature and potentially long duration of service execution necessitates asynchronous invocation and collection of results. Novel needs and opportunities for optimization arise when composing megamodules. In order to meet these challenges, we have defined a purely compositional language called CHAIMS, and are now developing the architecture supporting this language. In this paper we describe CHAIMS and how it meets the challenges of composing megamodules.

1. Introduction

The way software is created is changing. There is a shift from programming software towards composing software out of existing and new components, and a shift from standalone applications towards distributed applications. Future software systems will be based more and more on the composition and integration of large and distributed software components. Such large components - we call them *megamodules* - will have a higher level of abstraction than traditional subroutines or object library components.

Functions only provide encapsulation of statements and expressions. Components based on the object-oriented paradigm have gone a step further: they encapsulate data and procedures [1]. Typical examples of object-oriented components are components for graphical user interfaces. These components and functions can be local or distributed. The most common way to invoke them is by using local or remote procedure calls. Applications using this kind of components have various common characteristics: The components and the client program using these components are written in the same language, or at least in languages on the same abstraction level. The components are operated and maintained together with the application using them. Also, the components are often created together, and they form a coherent library. The application and the components share a common ontology and computing infrastructure. In case of distributed components, one common distribution system is used, e.g. either DCE [2], CORBA [3], RMI [4], or DCOM [5].

Megamodules differ from these kind of components in various aspects. Since they are larger, and marketed as services by autonomous providers, we must assume that megamodules not only encapsulate data and procedures, they encapsulate data, behavior, knowledge, concurrency and ontology [6]. Examples are reservation systems or transportation systems, that are maintained and operated autonomously by the service enterprises. They usually support multiple concurrent activities. The infrastructure in which they offer their services, and the concrete interface these services provide, differs among such megamodules, and cannot be controlled by those wishing to use these services. The composition of suc h megamodules contains various challenges: The composition must be able to cross the boundaries of languages, of distribution systems and of ontologies. The person doing the composition should be mainly a domain expert and not necessarily a computer expert in distributed systems. The concurrent nature of megamodules has to be accounted for, and for larger systems, compile- as well as run-time support for finding the optimal concurrent execution becomes necessary.

One way to meet these challenges is by having a compositional language, or *megaprogramming language*, which is on a higher level of abstraction than traditional programming languages. In the CHAIMS project (Compiling High-level Access Interfaces for Multi-site Software) we have developed such a compositional language, which on the other hand is not intended for computational use. We are also building a middleware system that supports composition by the CHAIMS language and uses existing middleware systems like CORBA, DCE, RMI and DCOM. The CHAIMS

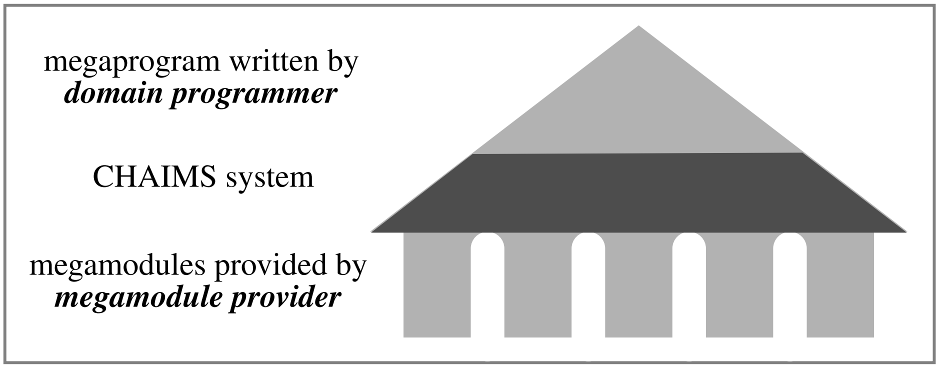

Figure 1. Separating domain and technical knowledge

environment, consisting mainly of a compiler for compiling CHAIMS megaprograms and of wrapping tools for wrapping non-CHAIMS compliant megamodules is tailored for the large-scale composition of big megamodules in a heterogeneous environment. In section 2 of this paper we describe into more detail the challenges of composing megamodules and our answers to them in the context of CHAIMS. Section 3 presents the architecture of the CHAIMS system, its run-time infrastructure as well as its composition environment. Section 4 discusses possible application domains for CHAIMS, and section 5 finally outlines future work and summarize early results of our research.

2. Challenges and solutions

In this chapter we discuss various challenges that need to be met when creating middleware systems for composing megamodules, and we describe the approaches chosen by CHAIMS.

2.1. Make composition available to non-technical domain experts

We assume that megamodules are autonomously managed and operated. Though they must be made public in an agreed-upon manner, the various megamodules are not provided by the same set of people. Also, quite naturally, the people interested in composing these megamodules - the *domain programmers* - most often will not be the same as the ones providing the megamodules. In contrast to the domain programmer, the *megamodule providers* need to be knowledgeable about the specific technical distribution protocol used to export services, and must have the technical skills to program and provide megamodules. But it is unreasonable to put this requirement on the domain programmers, who must focus on having domain knowledge. The challenge is to free the domain programmer from any knowledge of distributed systems and computational programming. Domain programmers need only be experts in composition and know the services required in their domain of application.

We therefore distinguish between two main actors in the composition process, the megamodule provider and the domain programmer (see Figure 1). We assume that these two roles are occupied by different persons with differing skills and objectives. Megamodule providers are classical programmers who write new or wrap existing megamodules for certain problem domains in order to make them available for domain programmers. The domain programmers are the ones who are primarily knowledgeable about the problem at hand. They have access to the documentation about available megamodules and have been trained in megaprogramming. Knowledge about distribution systems or experience in traditional programming languages are not prerequisites for them. The megaprogramming language we explore hides any details of the various distribution protocols used for communication, and it obviates the necessity to write in a common programming language. These tasks are taken care of by the composition system.

2.2. Support concurrency by asynchronous service calls

Megamodules are distributed and therefore can operate in parallel. Megamodules may be of substantial size, and the invocation of their services may take a long time. Thus it becomes imperative to take advantage of the inherent parallelism among megamodules. We achieve concurrent execution of megamodules by moving away from synchronous method invocations towards asynchronous method invocations. Several methods can be invoked in parallel, and the results of these invocations are extracted only when needed for further execution. Thus synchronization only takes place when necessary. This is also reflected in the megaprogramming language we have designed: The language primitive INVOKE actually only initiates a remote method execution. EXAMINE looks at the state of a currently running invocation and reports back

Pre-invocation:

SETUP, setting up connection to megamodule
SET-, GETATTRIBUTES, setting global parameters in megamodule
ESTIMATE, getting estimate of execution time for optimization

Invocation and result gathering:

INVOKE, invoke a specific method
EXAMINE, test status of an invoked method
EXTRACT, extract results from an invoked method

Termination:

TERMINATE, terminate method invocation or connection to megamodule

Control:

WHILE, IF

Figure 2. The primitives of the CHAIMS language

its status to the megaprogram. The EXTRACT primitive is used to get results of an invocation from a megamodule.

2.3. Need for novel ways of optimization and additional control

In a widely distributed environment, the availability of megamodules and the allocation of resources they need is beyond the control of the domain programmer. The challenge increases as megamodules become larger and more resource intensive. Furthermore, several megamodules may offer the same functionality. 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 is best done at run-time, as the compile-time estimation may change by the time the megaprogram is executed.

The domain programmer may also wish to dynamically adjust the performance of various megamodules by optimizing various setup parameters, e.g., search parameters or simulation parameters. Such parameters influence the speed and quality of the results, and the domain programmer may need to try several settings and retrieve overview results before deciding on the final parameter settings. This is also done best during run-time execution.

We meet the challenge of giving the domain programmer and the CHAIMS compiler the ability to schedule and plan for the execution of remote services by introducing an ESTIMATE primitive into the megaprogramming language. This primitive obtains pre-invocation estimates on the expected execution time. Scheduling tasks are to be taken over by the CHAIMS compiler. The compiler should optimize the execution order of the CHAIMS primitive in order to minimize delays. We also intend to introduce the possibility for automatic optimization of data flows between megamodules by providing direct data flows between megamodules. Furthermore, we plan to enhance the primitive EXTRACT for getting rough overview results from megamodules which will then determine further invocation parameters. This will also help to optimize execution time and the volume of data flow.

Traditional optimization methods have focused, quite successfully, on compile-time optimization. The CHAIMS primitives and architecture have the potential to support both run-time and compile-time optimization, thus conforming to the dynamic nature of a distributed environment. Here we intend to extend the work done on dynamic query optimization for databases into software composition [7].

2.4. Moving on to the higher level of a purely compositional language

In order to meet the challenges we have mentioned so far, the megaprogramming language we use for composing megamodules is purely compositional. In contrast to client applications that use (remote) components for certain parts of a program and implement other parts with a traditional programming language in the same client program, in CHAIMS all the computation takes place in megamodules. The megaprogram only composes megamodules, and the megaprogramming language only offers primitives for composition (see Figures 2 and 3). Yet as the language only focuses on composing megamodules, it can afford to offer new and more powerful ways for interacting with megamodules and invoking their services.

The CHAIMS language does not have a single equivalent to the traditional statements for synchronous invocation of a function or method (CALL statement) as found in most procedural and object-oriented languages. These statements are well suited for calling functions and methods within the same program in a synchronous environment. Yet when moving to large-scale distributed programming, too many diverse tasks need to be carried

```
// Megaprogram for finding the cheapest route for transporting certain goods
// from one city to another one

// bind to the megamodules
  io_mmh = SETUP ("InputOutput")
  route_mmh = SETUP ("RouteInfo")
  cost_mmh = SETUP ("AirGround")
  best_mmh = SETUP ("PickBest")
  best_mmh.SETATTRIBUTES (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_ih = io_mmh.INVOKE ("input_cities")
  input_goods_ih = io_mmh.INVOKE ("info_goods")
  WHILE (input_cities_ih.EXAMINE() != DONE) {}
  cities = input_cities_ih.EXTRACT()

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

  etc.
```

Figure 3. Example of a CHAIMS megaprogram

out by the CALL statement: handling the binding to a remote server, setting general parameters, invoking the method desired, and retrieving the results. By having separate primitives for these tasks we give the domain programmer and the compiler more control over the execution of the invocations. The primitives themselves can be synchronous since they don't induce computational delays. We now obtain the ability to introduce control over the timing of the invocations and can insert and rearrange primitives for optimization. Furthermore, having several primitives instead of one synchronous CALL statement gives us the necessary support for asynchronous calls of methods and concurrent execution of megamodules.

The CHAIMS language offers neither arithmetic operations nor any input or output functions. These are taken care of by megamodules, either by megamodules for mathematical functions as well as megamodules for input and output that are available as part of the CHAIMS environment, or by any customer provided megamodules offering such functions. These megamodules are used like any other megamodules.

The megaprogram itself does not inspect the contents of the data it receives by the EXTRACT primitive. It simply forwards this data to other megamodules for further processing or output. Thus, the paradigm of a purely compositional language leads to a clear separation between the data view and the composition view (see also Figure 4).

2.5. Heterogeneity

Megamodules may reside within different middleware systems. There are already a few dominant distribution protocols such as CORBA, DCOM, etc., and there is no reason to expect that one will dominate all the others. Furthermore, there is no reason to require that a domain programmer must know the details of such protocols. As such, it is necessary for a megaprogramming environment to interface to heterogeneous protocols and to provide a means of transferring data transparently between them. In order to facilitate this, the CHAIMS compiler generates client code for the different distribution protocols that the particular servers need. The CHAIMS language does not distinguish between the different protocols; any need to separate features is done by the system and is hidden from the domain programmer.

Wrappers for distributed software execution were developed earlier in the Polylith project, but that work did not have the benefit of standard communication protocols, so that it was harder to accomodate in a general and high-level framework [8].

2.6. Differing ontologies

We stated in the introduction that megamodules are also encapsulations for knowledge and ontology. When megamodules exchange data, this data may convey the same information, but the ontology they use to describe it may be different. When composing megamodules having different ontologies, this necessitates that the data exchanged between these megamodules be converted from one ontology into another one. A separate research project is exploring ways to meet this challenge [9]. In the CHAIMS project we do not yet deal with this challenge, instead we simply assume that those megamodules that are meant to exchange data also have matching ontologies, so that the CHAIMS system only has to deal with transforming data across language and distribution system boundaries, but not across ontological boundaries.

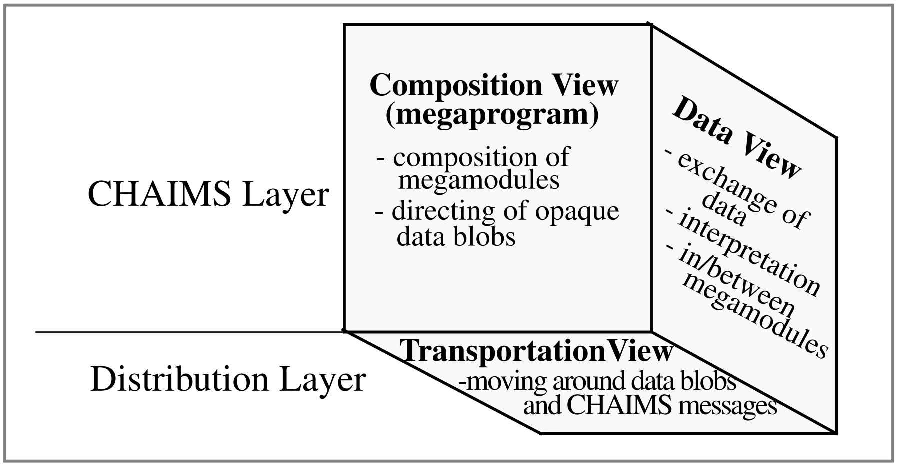

Figure 4. Composition, data and transportation views of the CHAIMS system

3. The CHAIMS environment

3.1. Composition view, data view, transportation view

In CHAIMS we make a clear distinction between the composition view, the data view and the transportation view. The transportation view is part of the distribution layer, the composition and the data views make up the CHAIMS layer which is on top of the distribution layer (see Figure 4).

The composition view deals with the composition of the megamodules as reflected in the CHAIMS megaprogram. The composition view is only concerned with controlling the requests to megamodules (determining the order of requests, invoking requests, checking execution, getting results). It is not concerned with the creation, modification and interpretation of data transferred between megamodules. The composition view is the only view seen by the domain specialist writing a CHAIMS megaprogram.

The data transferred between megamodules is represented in the data view. The data view describes the semantics of the transferred data as well as its encoding. The choice of a particular encoding protocol for computational data is independent of the composition view and the transport view. Changing the protocol only affects the data view, i.e., those parts of the megamodules that interpret the incoming data and create the outgoing data. Thus the data view and the composition view are orthogonal to each other. The protocol for the data view we have chosen is ASN.1, with the BER encoding rules for data transfer [10]. Other protocols would have been feasible as well. We refer to the ASN.1/BER encoded data as data blobs, since for the megaprogram they are just binary large objects.

The composition view and the data view make up the CHAIMS layer. The CHAIMS layer sits above the distribution layer which contains the transportation view. In the transportation view we use one or several distribution protocols, e.g. CORBA, DCE, DCOM or RMI. The transportation view is only concerned with the transport of messages between the megamodules and the megaprogram. The content of these messages is given by the composition view and the data view. The transportation view is neither concerned with the correct use of the CHAIMS primitives nor with the content and encoding of any computational data.

3.2. Components of the CHAIMS architecture

Figure 5 shows the architecture of CHAIMS. At the heart of the run-time system is the Client Side Run Time (CSRT). It consists of the compiled megaprogram and necessary stubs. This CSRT controls, through the CHAIMS primitives, the execution of the various megamodules and facilitates the exchange of data among the megamodules. The communication between the CSRT and the megamodules is done via various distribution layer protocols, such as CORBA, DCE, RMI etc.

The generation of the stubs for these protocols is taken over by the CHAIMS compiler when compiling a megaprogram. The CHAIMS compiler is part of the composition environment. Another component of the composition environment is the wrapper tool that automates the wrapping of megamodules which do not support the CHAIMS primitives (i.e., legacy megamodules). Finally, there will also be a repository where the megamodule providers can advertise their services. This repository provides useful information to the domain programmer and the CHAMS compiler during the composition process. We discuss in detail various components of the CHAIMS architecture below.

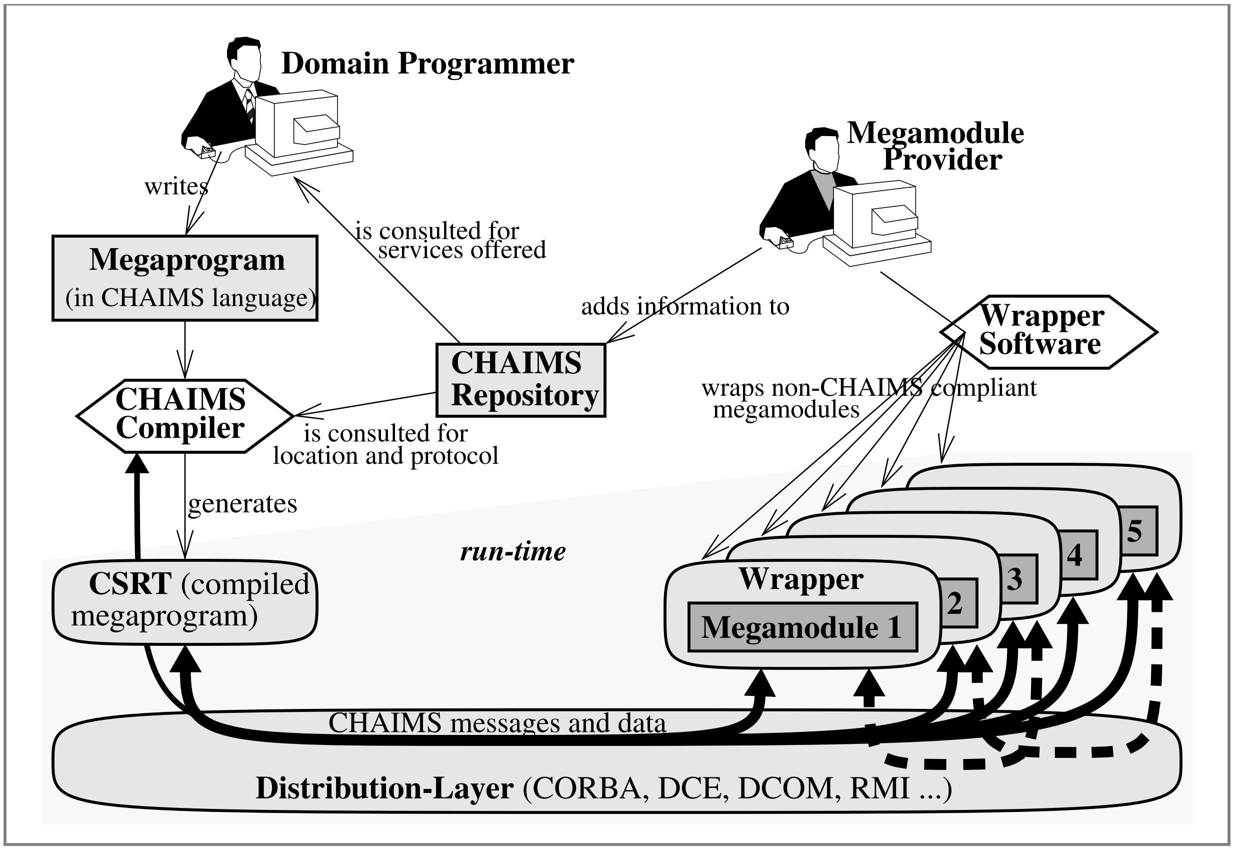

Figure 5. Architecture of CHAIMS

Client Side Run Time (CSRT). The CSRT consists of the compiled megaprogram, this includes all the necessary client stubs. The CSRT controls the execution of various megamodules by issuing CHAIMS messages. The exchange of data among the megamodules is currently done through the CSRT. The CSRT gets data in form of the ASN.1 encoded data blobs from one megamodule and passes that data on to another megamodule. It does not try to interpret or change the data. Thus the job of the client side is restricted to control and communication. The computation is delegated completely to the megamodules. This feature enables us to keep the CHAIMS language and the CSRT simple and lightweight, which simplifies the job of the domain programmer considerably. It will also enable eventual dataflow optimization.

Another innovation of CHAIMS is the concept of run-time optimization done by the CSRT. As explained in section 2, dynamic optimization is necessary in an environment consisting of large-size megamodules with varying availability. Run-time optimization is obtained by using the ESTIMATE primitive.

Megamodules. All the computation (including input and output as well as data retrieval and storage) takes place in the megamodules. The communication and execution control is done by the CSRT which issues various CHAIMS messages to the participating megamodules. However, megamodules derived from legacy systems will not understand the CHAIMS protocols. Such megamodules need to be wrapped. The wrapper performs the translation from the CHAIMS primitives to the interface understood by the megamodule, and vice versa. The wrappers also take care of the encoding and decoding of the ASN.1 data-blobs.

Translation is not the only functionality that the wrappers have to perform. As mentioned above, CHAIMS supports asynchronous invocation. Most existing megamodules only support synchronous invocation. In such a case, the wrapper needs to make sure that the wrapped megamodule supports asynchronous invocation. This support for asynchrony also means that the wrapper needs to do state management and some concurrency control as various invocations of the same method may be active simultaneously in a single megamodule.

The wrappers can be either manually coded or generated automatically. Currently, we support wrappers for the CORBA environment. These wrappers are primarily generated automatically and require only a little hand crafting. We aim to automate the wrapper generation as far as possible. This will make the introduction of a new type of megamodules into the CHAIMS system relatively easy. If a new megamodule uses a distribution layer protocol which is unsupported in CHAIMS environment, then we just need to generate wrapper for this type of megamodules and add the unsupported distribution layer protocol to the CHAIMS compiler so that corresponding client stubs can be generated in the CSRT.

CHAIMS compiler. The CHAIMS compiler generates a high-level language client program and the necessary stubs to communicate in the desired distribution protocol. For this it consults the repository to find the required information about the megamodules, such as their location, distribution layer protocol etc. The compiler will also be responsible for any compile-time optimization like scheduling invocations and determining direct data flows. In case the compiler does compile-time optimization it may also contact megamodules directly in order to get the necessary estimates. Figure 5 shows exchange of data and CHAIMS messages between the megaprogram and the megamodules and also between the compiler and the megamodules.

CHAIMS repository. The CHAIMS repository will be the place where the megamodule providers advertise their services. It provides documentation of the functionality of the various megamodules to the domain programmers and enables them to pick the relevant megamodules and call the relevant functions in the megaprogram. The repository also provides information to the CHAIMS compiler.

Distribution layer. This is the communication layer that is used to transfer the CHAIMS messages and data between the megamodules and the CSRT. Some examples of the distribution layer protocols are DCE, CORBA, JAVA-RMI and DCOM. Currently, we use these protocols only for the reliable transfer of the CHAIMS messages and data, and do not take advantage of the various other features such as security, name service etc. We may leverage such features in the future. We do not go into the details of these protocols in this paper.

4. Application domains that will benefit from CHAIMS

It is too early to identify the domain in which compositional languages and systems like CHAIMS will first gain strength. Possible areas that are well suited are logistics and planning systems where the actual services and data are often external to the enterprise. Planning often involves unpredictable problems and scheduling. Also, in the planning domain there already exist many servers that provide certain information and are capable of transforming and computing data. The services of these servers need to be composed by domain specialists on the fly in order to gain the desired information - this composition must be easy and technical details must be automated. For services with long execution times, pre-invocation estimates and scheduling will be essential. As the problem in question may only become clear after several iterations, optimizations concerning overview results could play an important role. Composition languages and systems like CHAIMS could prove to be a suitable approach to composition in this domain.

Large-scale composition as it is done by CHAIMS could also influence the way how we deploy software. Nowadays, many large systems are developed within a complex and sophisticated development environment that allows monitoring and quality control. Events and thus also errors can be traced, performance is tracked, and in case of malfunctions or crashes a quick recovery is made possible by those knowing the software and its development environment. Yet when such a software package is deployed and transferred to the site of the customer, its development environment is left behind [11]. The paradigm of megaprogramming envisages a different approach: the services are no longer moved to the site of the customer, only a megaprogram controls operation at the client side. The megamodules remain at the site of the software provider within their development environment, and would be continuously monitored and maintained by their developers. This mode will also lead to new licensing schemes: services are purchased instead of software packages, and they may be payed for as used charges.

5. Conclusions

The CHAIMS project investigates a new programming paradigm: compositional programming based on megamodules. Given the divergence between the needs of large-scale compositional programming and the immediate needs of traditional programming languages and structures, we believe that megaprogramming will be necessary to handle certain aspects of composing heterogeneous, distributed and mainly large megamodules.

CHAIMS provides a step in abstraction that is comparable to the step of abstraction from assembler languages, where the programmer writes and optimizes machine specific code, to high-level languages. Furthermore, CHAIMS continues the shift from languages and systems aimed only at computation (e.g., assembler languages that only offer a goto but no concept of subroutines at all), to languages and systems that offer subroutine calls or remote procedure calls in addition to computational elements, to languages and systems that focus only on composition. Yet CHAIMS is not an automatic programming system. It relies on the knowledge provided by the domain programmer and the megamodule providers. We do not envisage that in large-scale system creation the application dependent knowledge can be delegated to automatic code generation.

So far, we have defined the CHAIMS megaprogramming language and the supporting architecture. The basic infrastructure of this architecture has been implemented and will us allow to explore, implement, and prove the various concepts for optimization. This will include compile time optimization concerning the scheduling of CHAIMS primitives, optimization of data flows by having a direct exchange of data between megamodules, and allowing the extraction of overview results. Future work will also include implementing the repository and providing better support for the domain programmer. Furthermore the work of the domain programmer could conceivably be supported by a browser for the CHAIMS repository or by an interactive programming tool that itself can establish connections to megamodules and execute primitives.

CHAIMS is being evolved in an experimental research project at Stanford University. It is supported by ARPA order D884 under the ISO EDCS program, as well as by Siemens Corporate Research, Princeton, NJ. More information about CHAIMS can be found under the URL http://www-db.stanford.edu/CHAIMS.

References

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

[2] W. Rosenberry, D. Kenney and G. Fisher: "Understanding DCE"; O'Reilly, 1994

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

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

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

[6] G. Wiederhold, P. Wegner and S. Ceri: "Towards Megaprogramming: A Paradigm for Component-Based Programming"; Communications of the ACM, 1992(11): p.89-99

[7] 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

[8] J. Callahan and J. Purtilo: "A packaging system for heterogeneous execution environments"; IEEE Transactions on Software Engineering, vol. 17, 1991, pp. 626-635

[9] J. Jannink, S. Pichai, D. Verheijen and G. Wiederhold: "Encapsulation and Composition of Ontologies", submitted 1998

[10] "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

[11] L. Osterweil: Presentation at the EDCS Dry Run PI Meeting in Los Angeles, March 1998