ABSTRACT

Creating and managing large-scale software remains a task which requires many levels of expertise, well-defined processes, adherence to standards, and careful documentation. Even when all these pre-requisites are in place overruns and failures are common. We support an alternate paradigm to conventional concepts of software creation: composition. We have seen composition already being used in practice for a long time, but typically in an ad-hoc fashion by experienced personnel. Composition is becoming more common, as large-scale services, as databases, mathematical modeling tools, and web-browsers are combined to rapidly create substantial, but hard to maintain and often short-lived systems. However tools to support this shift are minimal and fragmented.

To address this issue the CHAIMS project is developing an approach based on a very-high level programming language (CLAM) for software service composition. This language supports the concept of mega-programming, that is programming by composition of remote services, typically provided by autonomous suppliers. Its compiler is starting to generate a variety of invocation sequences for current and developing standards for software interoperation, as CORBA and JAVA RMI.

CHAIMS is intended for building applications using large services, which we refer to as services. The size of service we envisage typically justifies a dedicated processor, although services can share a single processor when performance demands permit. A repository links suppliers of services and composers of applications, giving details of location, protocols, and data-types. In this paper we focus on the composition of heterogeneous services.

Keywords: Software Composition, Distributed Computation, Software Reuse.

1. INTRODUCTION

Large software systems can no longer be built in a timely manner by collecting requirements, analysis, and then partitioning the pieces to a myriad of programmers, and finally integration and testing. We hypothesize that after the Y2K effort few large software applications will be written from the ground up. Instead large system will typically be composed using libraries and existing *legacy* code.

Composition programmers will use tools that differ from tools used by base programmers. [Belady:91]. In composition existing resources are catalogued, assessed, and selected, and systems are assembled by writing glue code to combine them. If the resources are distributed the glue incorporates transmission protocols for control and data. Considerable expertise is needed for success: the composer has to understand the application domain, judge to what extent the requirements of the customers can be covered from existing resources, and often negotiate compromises. And then the composer has to understand and manage an overwhelming level of details of interfaces, options in the available resources, transmission protocols, and scheduling options.

CHAIMS intends to fill the gap in composition tools. By providing a language to support composition, CHAIMS supports a paradigm shift that is already occurring in industry: a move from coding as the focus of programming to a focus on composition. This shift may be invisible to many enterprises and educators, since there is no clear boundary in moving from subroutine usage to remote service invocation. There are hence few tools and inadequate education to deal with this change.

1.1 BACKGROUND

Ten years ago, composition of large-scale software was performed by experienced groups in large companies, as in IBM, Unisys, Fujitsu, Arthur Andersen and the other

‘Big Five’, and system contractors as SAIC, Lockheed, MITRE, Lincoln Labs, etc. Today, software composition has moved to UNIX and PC platforms, and an increasing fraction of the software workforce is engaged in composition, albeit without focused tools.

Several general-purpose programming languages have had some composition facilities included within their basic capabilities. Examples are the LEAP feature in SAIL, services in PLITS, the Courier Protocol in Interlisp, rendezvous in Ada, and tasks in PL/1 In the end, these facilities were unwieldy and did not enter common usage. The complexity introduced by these features may even have contributed to their decay. Some information systems have had extensions to manage distribution, as databases (SQL) and some report generators.

1.2 ARCHITECTURE

The components of CHAIMS are symbolically depicted in Fig. 1. The topmost component is the client, represented by a program that defines the composition. The client program only manages the invocation of the services, according to precedence constraints that typically represent dataflow dependencies. The client program is compiled by the CHAIMS compiler [PerrochonWB:97]. The resulting executable runtime program consists mainly of service invocations, expressed using the interface standard protocols that are appropriate for this service. Many of these invocations will involve remote accesses, since we assume that the services reside on multiple sites, connected by a communication network. Any needed datatype or computational conversions to be performed in the dataflow are invoked externally to the client program, assuring that the client program is a clean representation of the architectural intent for this application. The actual services are either pre-existing and wrapped to serve CHAIMS primitives or written to order in a suitable programming language.

1.3 OBJECTIVES

We observe that large-scale composition requires functionalities not available in current mainstream programming languages. Connections to remote sites have to be set-up, scheduling of computations that can operate on parallel has to be managed, data have to be shipped among programs that are written in diverse languages, and a variety of transmission protocols have to be managed.

1. By defining a high-level language (CLAM) in which an application specialist can compose resources, we expect that the issues involved in composition will be clarified. By actually having a compiler for the language, we are able to test and assess the concepts needed.

2. To support the composition concept we need to define a protocol for communication that can be driven from a high-level language. To broaden the range of resources, our CPAM protocol can access services using any of a variety of existing interoperation standards, exploiting ongoing work in client-server technology. Linking to actual remote computational resources we can demonstrate the concepts to a broad audience and provide guidelines to deal with this paradigm change.

3. We also have to consider the roles that people play at the various layers of the system. In common with any large-scale organization we must recognize specialized roles. These roles are seen to be long term, since the large-scale software will also be long-lived, so that adaptation and maintenance is more important than the initial creation of a system.

The CHAIMS project has a limited scope: there is no plan for automatic programming, CLAM supports only a few data types, and the CHAIMS environment uses many defaults rather than give the programmer a rich palette of choices in the invocation of software services. By focusing CHAIMS on interoperation in a multi-site environment, rather than on platform-specific code, we gain high-level support of megaprogramming concepts.

We will now now deal with these three issues in turn.

2. LANGUAGE

The CHAIMS megaprogramming language serves only service composition and scheduling. Its narrow focus should allow it to remain simple, although some significant new concepts are introduced. By isolating the concepts related to composition and thus reducing our conceptual scope, we can address the specific issues within a research project of modest size.

Since CHAIMS does not provide for automatic programming or knowledge-based techniques, the composer must know which services offer what functionality and must have access to a well-maintained CHAIMS repository. The process of CHAIMS client program generation and use is illustrated in Fig. 2. The composer builds the source code of the client program in the composition environment. During that process the programmer will access service descriptions in the repository and also obtain initial feedback from the CHAIMS compiler.

2.1 PRIMITIVES

The statements in the CHAIMS language fall into three categories:

1. Initialization and termination statements, which control connections to the services: SETUP, GETPARAM, SETPARAM, TERMINATE, and TERMINATEALL
2. Service invocation statements, which cause processing results or intermediate results for control to be generated: INVOKE, ESTIMATE, EXAMINE, and EXTRACT
3. Statements to control the flow of the client program: WHILE, IF .. THEN .., ELSE..

The specific functions of these statements are sketched in Section 3.1, and an illustration of a simple program written in CLAM is given in Figure 3. CLAM places some logical and data dependency constraints on the execution sequence of these primitives. The major logical constraint is that each service being invoked requires a SETUP. An identifier generated by SETUP is used for all subsequent accesses to this service. INVOKE, starting the execution of a service method, also generates an identifier, and that identifier is needed for EXAMINE and EXTRACT statements to check and return results.

The services being invoked are autonomous and independent from each other, but must individually support the CHAIMS protocol, CPAM. This allows suppliers to develop new services, to be easily accessed by CHAIMS programs. Such services can be programmed in the most suitable language and use the best representation for the given sub-task and solution approach. However, in the beginning there will be little compliant software available, so that we provide wrapper templates for legacy software to satisfy CPAM.

CLAM supports only a few datatypes, for two distinct functions:

1. For control of flow CLAM supports identifiers, integers, boolean, real, string, and a date type. The identifier type provides the handles needed for services and service invocations. Computations on the other types are performed using a CHAIMS supplied native service module. This module follows JAVA standards.
2. For processing data CLAM supports a bag, which holds a complex object type, using the ASN.1 standard. The CLAM program cannot discern the content of a bag, it can only move it between services, including native services for input and output.

The native service modules, supplied with the CHAIMS system, appear identical to remote service modules, so that their behavior can be replaced. Such replacements may be needed if the client program lives in an unusual environment, say an embedded system.

The ASN.1 standard supports a recursive data structure of triplets, where each data element consists of a descriptive name, type information and a value. The value is either again a triplet or a simple type, like bit-map, array of bytes, date-time, string, real etc. The wrappers convert this general object format into the local service formats.

The CHAIMS compiler is not burdened with having to deal with many types, any type conversions, or with code for fancy end user input/output. Interaction with the customer running the client program is dealt with through the native or substituted input/output services.

The binding of client programs to services can be delayed arbitrarily, and changed later by recompilation of the CHAIMS program. Delayed binding simplifies maintenance, adaptation to changing network conditions, and changes in service capabilities and costs. Delayed binding also relaxes commitments to specific interface standards. Choices among standard interfaces as CORBA, ActiveX, JavaBeans, and DCE can be deferred until the scale of the problem being addressed is known and the platform capabilities can be assessed. Even new interface standards can easily be adapted by updating the compiler, its library and recompilation of existing CHAIMS client programs.

2.2 CLARITY

The structure of a client program defines the architecture of a system, independent of its implementation. In effect, the model architecture will be clearly visible, and is not buried in interface and application-specific code. The client program defines an architectural instance of an application, while delegating all computational activities to the services it invokes. The architecture instance defined in a client program is reusable, not as a paper document, but when linked and recompiled with appropriate services, as a complete re-instantiation of the domain architecture. Since the compilation can bind to alternate interface specification languages and to alternate services, scalability and platform independence can be achieved, as long as equally competent services can be acquired. The client program will be limited by the capabilities of current interface languages, but these are improving rapidly.

We intend to provide a graphic editor to manage a graphic description of the client program's architecture. Here we do not need to be novel. Many tools for graphic displays of computer architectures are available today, although they may not drive actual system execution. Any reasoning tools they provide can provide assistance of course, although we hope that the client programs will remain of modest size, and CLAM discourages complexity. Final validation of a CHAIMS architecture instance is through its execution, including the execution of the client program in smaller environments.

2.3 PARALLEL COMPUTATION

Modeling the activity of real world objects, parallel operation of the services is the underlying assumption in CHAIMS. This vision means that the increasing availability of distributed computing can be exploited without resorting to parallel computing features applied to sequential programming languages. By considering any sequential dependency as an exceptional constraint, the composer will naturally think of parallel execution. This concept of natural parallelism is the alternative to the common paradigm seen today, where the programmer codes the actions to be performed in a world where everything lives in parallel into a sequential format, which is subsequently analyzed by parallelization tools to extract possibilities for parallel execution. Since activities in a natural, distributed world actually occur in parallel, we hope that CHAIMS client programs can capture their essential parallelism without dual translations.

CLAM achieves new flexibility in service scheduling by having split the functions of the traditional CALL statement into units that can be scheduled independently: SETUP, INVOKE, and EXTRACT. The ESTIMATE and EXAMINE statements, described in more detail below, provide the information needed for their scheduling. In addition to exploiting parallel execution of services, the communication bandwidth requirement among them can also be significantly reduced. Documentation of the optimizations will be covered in future papers.

2.4 SUMMARY

The focus of the megaprogramming language CLAM and the CHAIMS interface drivers is to reduce the cost of long-term maintenance and software evolution, and reduce the numbers of errors occurring in this process, without reducing the extent of maintenance and evolution actually being performed. Since CLAM only serves service composition and scheduling, client programs remain relatively simple, although efficient distributed computation is supported.

3. INVOCATION PROTOCOL CPAM

CPAM (CHAIMS Protocol for Autonomous Services) is our protocol for accessing and using the methods offered by services. Services are offering their services independent of a specific client. A client does not have to own a service, it just can use remote services, offered as methods to be invoked. The effect is that CPAM allows process composition.

CPAM has characteristics that address specifically the composition and reuse of autonomous, mostly distributed and computation intensive services [BeringerWL:98]. Having distinctive calls allows for the client to be simple while exploiting the parallelism of methods invoked from different services. We will briefly describe the CPAM primitives, and then explain the benefits of their separation.

3.1 CPAM PRIMITIVES

We briefly summarize the primitives here, more detail is provided in [SampleBMW:99].

Establishing a connection to a service: The primitives SETUP and TERMINATEALL establish and end the connection of a client to a service.

Cost estimation: ESTIMATE allows a client to ask a service for cost estimates for a specific method. It is available prior and during execution. The output is to be a name-tuple list (name of the cost factor, value of the cost factor and its uncertainty). If the service cannot provide estimates, its wrapper returns as reasonable values as possible, perhaps based on past executions. Cost estimates allow the composer or a CLAM optimizer to choose among alternative services and/or optimal execution paths.

Executing service methods: Methods are executed by the following four calls: INVOKE, EXAMINE, EXTRACT and TERMINATE. INVOKE starts the execution of a method, which then proceeds asynchronously, multiple INVOKES with different parameters can occur within a SETUP. EXAMINE reports the status of the execution, EXTRACT returns selected results, and TERMINATE deletes the invocation, but does not disable access to the service.

Presetting of attributes: The call SETPARAM is used to set default values for invocation attributes and global variables in a client-specific way. The complementary call GETPARAM simply allows checking of default values in a service or values set by SETPARAM.

3.2 CPAM EXECUTION ALTERNATIVES

Figure 4 shows the paths of execution that are possible with a single service, and an arbitrary number of such paths can be interleaved. The protocol allows exploitation of a variety of conditions with a single scheme:

1. As expected, multiple services can be SETUP in parallel, and their invocations interleaved. If useful, a single service can be SETUP multiple times, for alternate conditions.
2. SETUPs can be performed early and in parallel, so that subsequent data-constrained sequential INVOKEs can be rapidly executed. In many distributed computations SETUP can take more time than method invocation.
3. EXAMINE permits traditional polling prior to EXTRACT, allowing the client to overlap execution of INVOKES. Nothing novel here, but
4. EXAMINE can return progress indications from simulations and similar long-running or continuous executions (say weather predictions), so that the client can balance precision and result delivery time.
5. Multiple INVOKES to a service method can be made in parallel, typically with differing input parameters for the service, for instance to bracket a range of decision parameters.
6. A high uncertainty of costs of alternate optimization choices during compiling can be resolved by moving the ESTIMATE statement into the execution flow, when the values should have a lower variance.
7. ESTIMATE can be used to decide on scheduling or canceling of alternative services.
8. ESTIMATE can be used to make choices among service execution orderings, especially when pre-execution compilation can not resolve significant uncertainties.

9. EXTRACT can specify a few control values to be reported, to let the client program decide if a satisfactory solution has been obtained or more iterations of INVOKE with new input parameters are necessary.
10. Dynamic services can be interrogated using GETPARAM.

Many of these alternatives exist now somewhere, either as a feature of a specific programming system or through clever programming of an application. We do not know any approach which has provided the mix-and-match capability show in the CPAM protocol.

3.3 INTEROPERATION PROTOCOLS

CPAM is a protocol which is mapped by CHAIMS onto a variety of existing client-server protocols. We have used CORBA, JAVA RMI, DCI, and DCOM with various degrees of success and happiness. The strength of CPAM is however the flexibility of interface standard choice, changeable at any time by client program recompilation.

The size of the services we envision typically justifies a dedicated processor, although services can share a single processor when performance demands are modest. Services written in C, C++, Ada, FORTRAN, etc., will need interfaces (similar to APIs) to allow interoperation in the CHAIMS setting. Interoperation protocols from standards as CORBA, ActiveX, JavaBeans and DCE provide, in effect, our machine languages and make the CHAIMS concept implementable today. We believe we can succeed because we can build on these efforts being expended on interfaces for interoperation within client-server models. By moving up one level of abstraction in programming, we are moving to a new paradigm where the focus is on composition of services, rather than assembly of code.

3.4 ADAPTABILITY TO CHANGING STANDARDS

The move to composed software is clear, but still poorly focused. We now have as many proposed standards for software interoperation as we had computer hardware architectures thirty years ago. Modern programming languages now provide platform independence and programs can be recompiled for new hardware. The effect is that application software typically lives about 15 years, much longer than its hardware.

The same stability does not exist yet for interface software, especially when connecting computers of different types. For instance, SQL is undergoing changes in its transition to SQL-2 and -3. Selection and implementations of feature sets will vary widely. OMG's CORBA 2 will have many features not available in CORBA. KQML concepts are being suggested for future versions of CORBA , while JAVA is taking over server-managed client computing. The intent of XML is to replace HTML for processing applications, but will depend on an unknown variety of DTD definitions to achieve depth at the cost of consistency. New object-libraries arise, but are rarely compatible with those of other suppliers. We must surmise that interfaces for distributed computing are likely to stay fluid.

Just as traditional programming languages provide an insulation from platform differences, the essence of CHAIMS is to provide insulation for the composer from the differences in today's interoperation architectures and standards. Such independence is especially crucial for larger systems which need to operate on multiple sites, utilizing networks and a variety of services. These systems represent major investments, and have a long lifetime.

4. ROLE ASSIGNMENT

In a megaprogramming environment we distinguish two primary divisions, namely the servers and the clients roles. One application system is likely to use multiple servers, but be controlled by one client. Other clients can be composed of the same and other services. Figure 5 shows the main components of an instance of a CHAIMS system. The repository provides the means for sharing information about all the available services, and links supplier, composer, and end-user.

4.1 SERVICES

The concept of services implies autonomous ownership. The control over the component or service remains at the provider's site. This is a major distinction with traditional software models, where components or services, no matter how large, are subservient to the calling routine. A prime example of services today is provided by databases, but there are also some computational services available today. The Internet provides a wide variety of services, although they are rarely envisaged for composition, examples include weather services, airline ticketing, and book sales. Other potential services are simulation programs, design and construction programs, services for genomics [GennariCAM:98] [Searls:98] and for manufacturing [CramsieEa:97], business services [Oracle98]. Many more are expected to come into existence. But there exist yet few protocols supporting an integrated vision and allowing easy reuse and composition into a larger system.

The economics of a service are based on reuse. It becomes a benefit to the service provider to maintain the service. The rules and processes encoded in a service represent knowledge. This knowledge is subject to change, not just the data it operates on. As the world changes, recourses broaden and algorithms improve. Bringing this updated knowledge to the client, either by reusable components or as updated concepts and requirements to be integrated and implemented by the customer into their programs, is cumbersome. Legal issues for service providers have been analyzed by [ChavezTW:98].

Time and fee can be estimated by the service, and can then be directly taken into account when calculating the cost of using a certain method. This is not true for the third cost factor. Though the amount of data flow can be estimated by the service, the effective cost is not the amount of data but the time the data needs to be transferred. This time is client specific because it not only depends on the amount of data but also on the capacity of the connection, i.e., quality of connections, distance, and traffic volume.

Note that for client process optimization a high precision in estimation is not crucial. The scheduling alternatives tend to be limited, and if the choices are close in costs, then the choice does not even matter. Variance is more of an issue, since a high variance means that more choices must be deferred to execution time.

4.2 COMPOSITION

The programmer who writes the client programs becomes a composer of services. The composer's tools are knowledge of an application domain, an understanding of services at a high level, the specific interface information captured in the repository, the CLAM compiler, and the wrapper templates [BeringerTJW:98]. The repository is currently a simple text file with a graphical browser. It contains a description of all available services, their methods, and their attributes. All the valid service, method and attribute names are posted in the repository. The repository is the only formal information flow necessary between providers, composers, the end-users or customers using the client programs, the compiler, and other tools in CHAIMS. The compiler compiles a client program written in the composition language CLAM into a client side run-time (CSRT), including the generation and compilation of all necessary stubs for various distribution systems. The wrapper templates are provided as part of the CHAIMS system to facilitate the wrapping of legacy services into CPAM compliant services.

The composition language CLAM together with the CHAIMS system disengages the domain experts from technical details like the use of complex programming languages and the programming of distribution systems. This can be compared to today's use of database management systems, where the SQL programmers are quite distinct from the programmers who work at the DBMS provider, and it is unlikely that they have ever met. We hence assume that these two roles are occupied by different persons with differing skills and objectives.

CHAIMS should make it easier to train domain specialists in composition rather than untrain conventional programmers that are used to all forms of restrictions. We still have to learn to what extent CLAM should not be similar to an existing sequential programming language, so that it will be clear to the composer that the environment is inherently parallel, and that much prior experience and training will have to be unlearned.

4.3 APPLICATION CUSTOMERS

The customers of the client-programs live in the same domain as the composers, so that interaction is easy, and adaptation to changing customers need is rapid. If the customer is technically confident, and the tools are reliable, the customer and the composer may be the same person. When new services are required, searches through the repositories are the first task. If no suitable service is found, then wrapping of a legacy service has to be negotiated.

The focus of CHAIMS is mainly on computational services, complementing the now dominant information services. Especially in the domain of web-information, information is downloaded from an information server, and then put manually (cut and paste plus maybe some cumbersome conversions) into another program that performs actual processing, say, a spreadsheet. Our demonstrations have focused on logistics and scheduling applications, since in that area suppliers of transport, warehousing, and breakdown services have a motivation to provide assistance to their customers, while the customers have many choices to assemble these services. Our case studies include a logistics example ("find the best route from city A to city B under certain circumstances") using several services for the various parts of the computation, and an aircraft design example with services for the computation of the structure, the control elements and the static of an aircraft wing.

5. CONCLUSIONS

We described a tool that supports a non-traditional, but realistic approach to software systems development. Rather than following the waterfall model or its variations by starting from specifications, through design, etc. to code generation, CHAIMS assumes that large programs can best compose from existing services. This limits the application client programs to the composition of available resources, or Megaprogramming [BoehmS:92]. The benefits of the approach are largely in terms of long-term maintenance of computer systems, which consume 70-90% of total systems costs.

In a megaprogramming approach, composers are willing to give up control for the benefit of expert maintenance at the source sites in a collaborative setting [WiederholdWC:92]. Megaprogramming distinguishes itself from database integration by incorporating knowledge embedded in programs, rather than being limited to declarative knowledge applied to databases. Database functionality can be incorporated into megaprogramming through server programs that execute SQL SELECT statements, but these languages - focusing on a single verb - are known to have inherently limited computational capabilities [Ullman:88].

Megaprogramming can also be viewed as large-scale object-oriented (OO) technology. OO increases the procedural capabilities of distributed objects [Booch:91], but is restricted in practice to single protocols and coherent libraries [AtkinsonBM:95]. The high-level language approach of CHAIMS scales the object-oriented paradigm to autonomous service objects.

ACKNOWLEDGMENTS

The concepts we have assembled at a high level have their origins in much related work. In this brief note we cannot cite all of them, but we wish to acknowledge them. Specific references are given in other papers describing our research. Motivation for the work was also provided by a business school study into modern software practices. The development of client-server protocols, and problems seen in the maintenance of these system provided additional motivation and insights.

The CHAIMS project is supported by DARPA order D884 under the ISO EDCS program, with the Rome Laboratories being the managing agent and by Siemens Corporate Research, Princeton, NJ. We thank Mehul Bastawala, Ron Burback, Jing Chen, Joshua Hui, Pankaj Jain, Wayne Lim, Lawrence Lo, Louis Perrochon, Woody Pollack, Gregory Rokita, Kumaran Santhanam, Guido Schryen, Catherine Tornabene, and Felix Yu for their earlier contributions to the CHAIMS project.

REFERENCES

[AtkinsonBM:95] M.P. Atkinson, V. Benzaken, and D. Maier (eds.): Persistent Object Systems: *Springer-Verlag and British Computer Society*, 1995.

[Belady:91] Laszlo A. Belady: "From Software Engineering to Knowledge Engineering: The Shape of the Software Industry in the 1990's"; *International Journal of Software Engineering and Knowledge Engineering*, Vol.1 No.1, 1991.

[BeringerTJW:98]. Dorothea Beringer, Catherine Tornabene, Pankaj Jain, Gio Wiederhold: "A Language and System for Composing Autonomous, Heterogeneous and Distributed Megamodules"; *Proc. of Data and Expert Systems (DEXA) Software Composition Workshop*, IEEE, August 1998, p.826-833.

[BeringerWL:98] Dorothea Beringer, Gio Wiederhold, Laurence Melloul: "A Reuse and Composition Protocol for Services"; *Proc. Symp. on Software Reusability (SSR'99)*, ACM SIGSOFT, Los Angeles CA, May 1999.

[BoehmS:92] B. Boehm and B. Scherlis: "Megaprogramming"; *Proc. DARPA Software Technology Conference 1992*, Los Angeles CA, April 28-30, Meridien Corp., Arlington VA 1992, pp 68-82.

[Booch:91], [Booch:94] Grady Booch: *Object-Oriented Design with Applications*, 2nd Ed.; Benjamin-Cummins, 1994.

[ChavezTW:98] Andra Chavez, Catherine Tornabene, and Gio Wiederhold: "Software Component Licensing Issues: A Primer"; *IEEE Software*, Vol.15 No.5, Sept-Oct 1998, pp.47-52.

[CramsieEa:97] Bill Cramsie et al: CIIMPLEX Reference Architecture; *Consortium for Integrated Manufacturing Protocols (CIIMPLEX)*, Atlanta GA

[GennariCAM:98] J. H. Gennari, H. Cheng, R. B. Altman, & M. A. Musen: "Reuse, CORBA, and Knowledge-Based Systems"; *Int. J. Human-Computer Sys.*, Vol.49 No.4, pp.523-546, 1998..

[Oracle98] "Oracle Business OnLine, Removing Barriers to Enterprise Applications Adoption", http://www.oracle.com/businessonline/, Oracle Corporation, 1998.

[PerrochonWB:97] Louis Perrochon, Gio Wiederhold, and Ron Burback: "A Compiler for Composition: CHAIMS"; *Fifth International Symposium on Assessment of Software Tools and Technologies (SAST97)*, Pittsburgh, 3-5 June, IEEE Computer Society, 1997, pages 44-51.

[SampleBMW:99]. Neal Sample, Dorothea Beringer, Laurence Melloul, and Gio Wiederhold: "CLAM: Composition Language for Autonomous Megamodules*"; Third International Conference on Coordination Models and Languages (Coord99)*, Amsterdam, The Netherlands, April 1999, to be published in a Springer LNCS volume.

[Searls:98]David Searls; "Biowidgets"; Computational Methods in Molecular Biology, Elsevier Science, 1998.

[Swenson98] Keith Swenson, "Simple Workflow Access Protocol (SWAP)", Internet-Draft submitted to *WfMC* (Workflow Management Coalition), available at http://www.ics.uci.edu/pub/ietf/swap/

[Ullman:88] J. D. Ullman: Principles of Database and Knowledge-Base Systems; Volume 1: *Classical Database Systems, Computer Science Press*, 1988.

[WiederholdWC:92] Wiederhold, P. Wegner and S. Ceri: "Towards Megaprogramming: A Paradigm for Component-Based Programming"; *Communications of the ACM*, 1992(11): p.89-99.

FIGURES

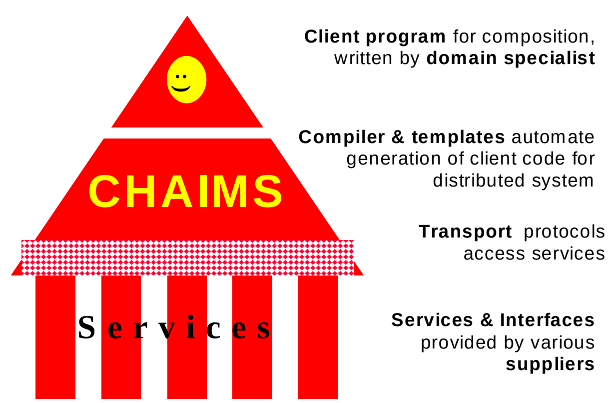

Fig. 1 Components of the CHAIMS megaprogramming paradigm

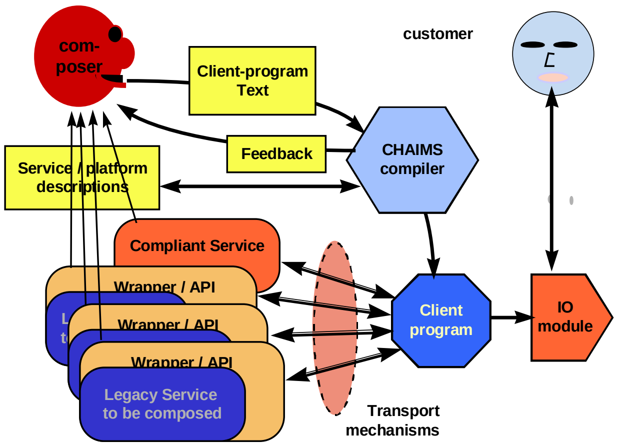

Fig. 2 The CHAIMS megaprogramming process and information flow

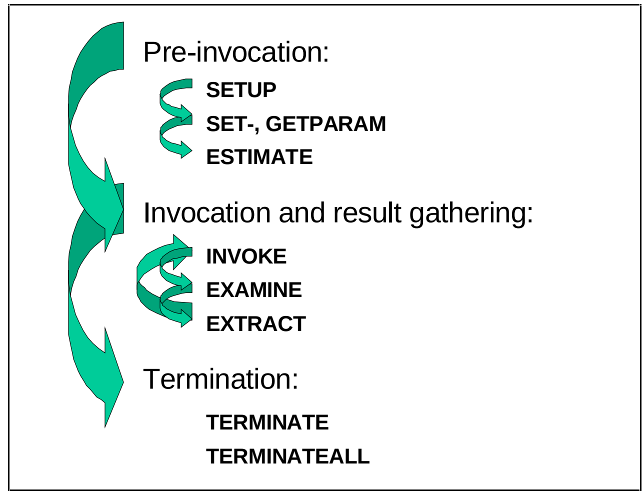

Fig. 4 Sequencing of CPAM primitives

```
BEGINCHAIMS
  io    = SETUP("io")
  math  = SETUP("MathMM")
  am    = SETUP("AirMM")
  gm    = SETUP("GroundMM")

  // Get source and destination cities
  ioask = io.INVOKE("ask", label="which cities")
  WHILE(ioask.EXAMINE() != DONE) {}
  (cities_var = Cities) = ioask.EXTRACT()

  // Calculate 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()
  gm.TERMINATE()
  io.TERMINATE()
  math.TERMINATE()
ENDCHAIMS
```

Fig. 3 CLAM program to calculate the best route between two cities

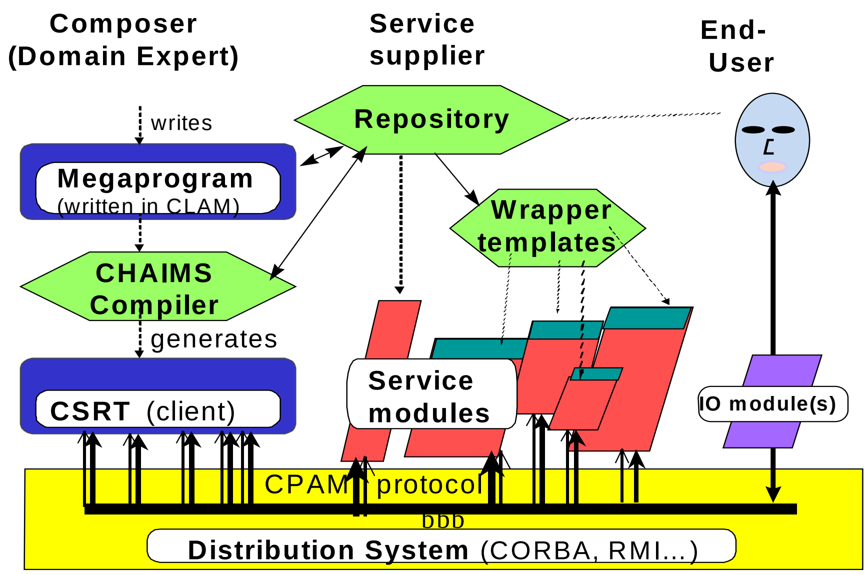

Fig. 5 Overview of the CHAIMS system