Applying Aspect-Orient Programming Concepts to a Component-based Programming Model

Thomas Eidson, Jack Dongarra and Victor Eijkhout

***Abstract*— The execution environments For scientific applications have evolved significantly over the years. Vector and parallel architectures have provided significantly faster computations. Cluster computers have reduced the cost of high-performance architectures. However, the software development environments have not keep pace. Object-oriented and component-based languages have not been widely adopted. Distributed computing on local area networks and Grids is only being used by a most number of applications.**

Clearly, there is a need for development environments that support the efficient creation of applications that use modern execution systems. This has been the goal of a continuing research effort over the last several years. The previous focus has been on using component-based ideas to develop a programming model and associated framework to support such a development approach. In this paper, two additional concepts are added to the base approach. Aspect-oriented concepts are applied to support the reduction of intertwined code related to different programming concerns; mixing I/O with a numerical computation is one example. Particularly in large applications, intertwining code can lead to applications that are difficult to modify and to manage. The second concept being added is the use of behavioral meta-data. When coupling smaller pieces of code (or components) to make a larger composite application, one needs to determine the suitability of the internal behavior of component as well as the compatibility of its interfaces. The objective is to integrate some of this information into the component and design a framework assist the programmer in making these decisions.

I. Objective of Target Framework

A. The Focus for the Programming Model

Programming efficiency has generally been a problem in the development of scientific applications. [1] [2] Application developers must juggle a number of concerns, which include the correct implementation of algorithms, high performance requirements, and use of large data sets. The need to analyze the output adds other programming aspects, such as the use of graphics routines. The ever changing architecture designs of computers further complicates the scenario. Also, target problems are transitioning from a narrow focus (i.e., the study of an isolated physical phenomenon) to multi-discipline applications where different element applications must interact to correctly capture the target problem. [3] [4] And, the need to share part or all of an application is increasing.

Clearly, the development of modular, well-organized applications that are portable is an important goal of most application developers. The use of library packages has traditionally been the means of supporting such a programming requirement. But, many applications still end up with the different types of programming aspects coded in an intertwined style which results in applications that are hard to decipher and to manage. The intertwining is done sometimes to improve the performance of applications, but also is done because traditional programming systems do not provide or encourage alternatives.

Humans are better able to understand and manage complex systems if they can be described as a set of individual concepts along with a set of coupling procedures. Object-oriented and component-based systems provide some support, but they still leave difficult implementation details to the programmer. A programming approach, referred to as aspect-oriented programming (AOP) is particularly focused on the programming of cross-cutting aspects of applications. [5] [6] An AOP-based system should encourage and make straightforward the coding of different programming aspects. The system should then provide support for weaving those aspects to generate a correct and efficient program. This can be done both during compile and execution phases.

In this paper, the primary concepts behind a proposed component-based framework for high-performance distributed scientific applications are discussed. These concepts are a result of years of experience in developing scientific applications and associated programming environments. [7] [8] [9] [4] [10] The framework being built uses a modular programming approach and uses meta-data to describe various aspects of the application design and thus compliment the primary algorithms that are described by traditional programming languages. The framework will provide support for weaving the separately coded aspects together to support the building of flexible, portable applications with good execution performance that model complex physical problems.

II. Definitions

1) A *framework* is an integrated collection of software tools that facilitates the development and execution of an application.
2) An *element application* is a code in stand-alone executable or library form, that is focused on a relatively narrow aspect of some physics, mathematics, graphics, or other science.
3) A *task* refers to a unique use of an element application. For example, each entry point into the code for an element application can define a different task. Also, configuration details for an application could be used to define different tasks that use the same entry point. However, different arguments passed to an entry point would not be considered as a new task.

Old Dominion University, Norfolk, VA
University of Tennessee, Knoxville TN 37996

4) A *context* is defined as a collection of tasks and data sets packaged for execution and interaction with each other. These tasks and data sets share a common address space. A Unix process is an example of a context.
5) A *platform* is one or more computers managed as a single entity that is connected via a network to other platforms. A collection of hardware with multiple CPUs that is managed as an unit with the intent of running the same code on all CPUs (generally, in a data-parallel style) is considered one platform.
6) A *composite application* is a collection of tasks and associated data that would benefit from being distributed among several contexts located on several platforms. The application typically includes a range of data and event transfer operations between the various tasks and contexts that make-up the application. For this discussion, a data-parallel code is considered as one task that runs in one multiply executed context on one multi-node platform.
7) A composite application is generally built by combining several element applications together under the control of a *work-flow* description. The work-flow code generally runs in a console or desktop environment associated with a framework.
8) A *software component* is a basic unit of software packaged for use in efficiently building some larger composite application. [11] The software package includes meta-data that defines any interfaces to that software so that some framework can more easily provide the necessary integration. Software component technology is intended
   - to support software reuse and sharing,
   - to simplify use of multiple languages,
   - to support the efficient building of large applications, and
   - to assist building distributed applications.

A. Target Applications

The proposed framework will target applications that potentially have a wide range of requirements. The focus will be on composite applications that couple element applications that require high-performance supercomputers with other element applications that runs on a workstations. So, while the details of developing individual elements are important, the proposed framework primarily addresses the integration of all these elements. An important emphasis will be on handling elements that are located across a heterogeneous network of computers. This includes the coupling of both loosely- and tightly-coupled element applications.

The target applications are assumed to include a significant number of executables, files, data transfers, events, computing resources and other entities that need programming control. For large applications, this means that a good organizational strategy is needed to manage all the configuration and control information that results in a correct execution. On the other hand, the basic approach must also support its use with small development applications. Large composite application will evolve from small element applications. The large overhead of modifying element applications has often been an obstacle to building composite applications.

III. The Programming Model

A. Programming Components and Meta-data

Most composite, distributed applications can be implemented via a set of software components that are controlled by remote requests from some work-flow program and by using the services provided by some framework. Complex applications will need to allow remote user code to also issue remote requests. The role of the framework is to manage the connections between the various local and remote components and to provide functionality that facilitates the programming and execution of the composite application.

The success of a programming model and an associated framework for developing such applications can vary depending on how well the framework matches the needs for complexity of the application. A good programming model needs to balance programming flexibility and efficiency with the application execution efficiency and accuracy. Programmers like for the programming abstractions to suggest good programming construction while also providing flexible control with a range of options. Most component-based programming models focus on the flexible management of software component (or task) executions. But in many distributed applications, other programming entities need to be managed besides the task programming entity.

Different types of programming entities might include contexts, platforms, data sets, events, files, messages, and other aspects of a code that define execution requirements other than the core computation aspects (i.e., the basic algorithm being modeled). The base idea in the proposed programming model is to treat these programming entities in a similar manner as the task programming entity is typically programmed in a component-based system. In the proposed model, a *Programming Component* is defined as a programming entity plus a set of meta-data along with a set of framework methods. [12] A Programming Component family is a set of Programming Components all with the same programming entity type. The *meta-data* is a set of configuration data that defines interfaces, parameters, macros, and separate aspect code related to the programming entity. *Framework methods* are a library of routines (functions, methods, subroutines) that are provided by the framework to perform operations specific to each Programming Component family. (Framework methods are discussed in more detail in the next section.) Generally, they use the meta-data rather than arguments to define specific functionality desired by a programmer. They can be implemented as direct calls to library routines or as services provided locally or remotely. The essential feature of a service is that it is executed as a separate computational thread. This allows for parallel execution to support performance needs but also allows a service thread to manage related aspects that are programmed in multiple places in an application.

B. Base Use of Programming Components

The original objective of the proposed programming model was to support the management of large composite applications. It evolved from a distributed application built several years ago named FIDO and was refined during the development of a subsequent prototype framework, LAWE. [4] [10] [8] The goal of the FIDO project was to a implement a multi-component application, the multidisciplinary design optimization of a specific airplane, into an integrated system that ran on a heterogeneous network. The number of programming entities of each type was of order 10. Configuration files were defined in an ad hoc manner to store similar types of information. The result was an organized system that was reasonable but that probably would not scale to more complex problems. Upon review of the project, it became clear that organizing the configuration data around specific programming entities resulted in an overall organizational structure that was efficient to program. The LAWE system was built around a meta-data database for Programming Components where the application developer defined specific Programming Components for a target application. The result was a clear and efficient description of a composite application.

The proposed approach supports the accurate programming of the various global programming entities. The application developer creates one or more definitions (or members) of each Programming Component family which are stored in a composite application database. Each individual definition (conceptually similar to a class object in object-oriented languages) has a global name to support its accurate use in different codes which could be written by different programmers. For example, a data set entity that is transferred between two task codes will have only one definition; that being the one in the database.

At runtime, the framework, the work-flow code, or even a user task code can create instances of each member of a Programming Component family. These instances have an identity that can be shared among the various user code to further support accurate programming. The framework will provide services to support the discovery of instances created by the framework or other tasks.

Figure 1 shows an example application with two task families. Task C runs in a console context (i.e., a process on the user's desktop) and Task R runs on a remote-server context (i.e., a process on some remote computer). Pseudo-code for Task C that defines the work-flow is shown in Figure 2. In this example, Task C creates two contexts, X and Y, each of which have been configured to include an instance of task R—R.1 and R.2. In this example, the instances of R are assumed to be created as part of the creation of the associated context and "discovery" methods are called in the work-flow that use an implicit instance identity—the assumption that only one instance of a task is available in each associated context. Otherwise, an instance identifier would need to be passed to the work-flow code. [X and Y may possibly execute on two different remote computers.] Task C is returned a handle to each instance of Task R. A handle is just a referencing variable that the framework library uses locally to define instance identity. Task C then repeatedly requests that R.1 and R.2 be executed concurrently inside a loop until the work-flow objective is satisfied. This example also shows a relatively simple strategy for programming multi-threaded distributed applications. Initiation of some "use" method for one or more instances of different Programming Components can be started concurrently. A "wait" procedure can be called at some later appropriate point before accessing the results of that use. Each use can be managed by the framework via a separate thread.

The focus on Programming Components allows the user code (or tasks) to be written to run with a high degree of flexibility because each code can contains a minimum of detail about the different programming entities or aspects of the application. Discovery methods allow accurate management of similar instances of application aspects. Use methods allow control of a programming entity to be coded without hard-coding details. Use methods can read the associated meta-data to provide specific functionality. The functionality can be easily modified by changing the meta-data during application development or dynamically during execution. Method arguments can even provide additional dynamic capability if that is needed. Example use methods are:

1) to start remote executables defined by a context,
2) to execute the code defined by a task that resides in a server defined by a context,
3) to copy files defined by a file entity,
4) to transfer a data set defined in the memory of a context, and
5) to check on the status of an event.

The approach is particularly useful for distributed applications. The framework methods can be used to directly control remote functionality in a task-parallel programming style. This will allow very complex applications to be developed using a manageable modular approach. In addition, the approach supports efficient execution implementations. Because the various programming entities are managed via framework methods and the framework understands the meaning of the meta-data, the framework can choose the best of competing implementations. Also, the framework methods can be implemented as filter and and service tasks. Filter tasks can be used to easily alter the effects of a user programmed framework method. A message transfer method might be coded in a task to transfer a large data set. A possible implementation is for the framework to provide a filter task that receives the address of the data and actually does the message passing. If all the data is not needed, then the filter task can be changed, outside the scope of the task code, to only send the needed data. Service tasks would be implemented using multi-threaded programming. They would allow a great deal of flexible, dynamic control of framework method functionality. The service task could even be configured in an interactive mode so that a user could alter behavior during execution.

The proposed programming model is targeted initially for implementation via text based languages. However, in the LAWE project a simple graphical programming system (using Java Beans) was developed to describe application work-

flow. [8] [13] [14] This prototype demonstrated the potential for a valuable approach to describing a composite application. Composite applications are often developed by teams; some of whom do not program. Historically, flow charts would need to be developed to describe the application to others. The combination of a visual work-flow description along with the meta-data database provides this capability without the extra work. Moreover, when the application is modified, this first-level documentation is automatically modified.

C. Aspect-Oriented Programming

One value in defining these Programming Components is to support the programming style of separation of concerns that is a primary principle of AOP. Separation of concerns refers to a programming style where the details of each programming concern (or aspect) are located together rather than being scattered throughout a code. This point would be of trivial importance except that the executions related to these details may need to be scattered throughout a code for correctness or performance reasons.

The base benefits described above still require some detail of various aspects to be located in the work-flow or task code since framework methods need to be explicitly called. And, this detail may need to scattered throughout the application code. Essentially, Programming Components describe and control side-effect behavior of task codes. In other words, they control behavior outside the scope of the task code that needs synchronizing with the execution of the task code. If one analyzes the primitive requirements of these side-effect aspects, they boil down to the following:

1) identification of data sets (memory locations) inside the scope of the task code;
2) knowing the current execution point (or region) in the task code; and
3) understanding the data dependencies associated with those data sets and the current execution point.

Generic framework methods can be defined which allow a programmer to annotate a task code with the above information rather than coding more specific Programming Component methods. This approach will provide a dramatic improvement in task code portability. It should be generally easy for a programmer to determine the needed generic methods while developing the task code. The identification of data sets that might be useful outside the scope of the task code should be generally straightforward. Eventually, associated compilers (and even loaders) can be designed to allow extraneous generic framework methods to be eliminated from compiled (and executable) code and the programmer can use a liberal approach when adding generic methods to define data sets. In other words, when the methods are not needed they can be easily removed. Data dependency behavior can be programmed by including lock and unlock methods. A programmer should know when a data set is being read or written and needs to be locked. Control point methods that have labels to identify execution points can be liberally included similar to the data set identifier methods. Coding generic functionality will provide more programming freedom as the developer will not have to decide on the specifics of side-effects until that code is actually coupled to other codes.

The more specific Programming Component methods can then be programmed in separate aspect codes, that use the control point labels and data set identifiers to describe the desired, correct execution. The framework will provide the weaver functionality to integrate the different aspects during execution.

Historically, this type of functionality has been provided by methods or functions provided by a library that is linked to the user code. The proposed model allows the functionality to be implemented with a service model. For example, file I/O is often programmed by coding an open statement along with a set of read or write statements that might be intertwined throughout a code. All this can be replaced by control points and the I/O is located in the aspect code that is implemented in a server style. This will allow for the I/O programming to be centralized for more efficient management of that code. For example, it may be desired for several tasks to write to the same file. In the proposed model, none of the tasks need to provide any file control, like open statements. This is configured via aspect code and implemented via a service task. Programming flexibility is also enhanced. Rather than write the data to a file, an alternate composite application may need to send the data via message passing. The propose approach allows this to be done easily by altering the separate aspect code. The scattered generic methods in the user task code would not need be changed.

D. Behavioral Analysis

a) Potential Use: The above discussion focuses on making the programming mechanics of developing and maintaining applications easier. As computational science evolves, more composite applications will be built from libraries and shared element applications. The application developer will not be intimately familiar with the details of the algorithm for every task. Even good documentation has never proved completely satisfactory, because the documentation writer cannot know the key details that each user wants highlighted. [15]

As algorithm disciplines mature, the description of the behavior of specific algorithms to solve specific classes of problems can be standardized. The knowledge and expertise of the algorithm developer can be conveyed through behavioral meta-data using standardized vocabularies. The application developer can then express the application needs via a similar concept—a contract for requirements of the application. The programming framework can then analyze the contract and the meta-data for target tasks to suggest good choices. Clearly, the sophistication of this analysis will be initially limited as the scientists in each discipline learn to quantify their expertise.

The use of behavioral meta-data does not have to be limited to a description of the numerical methods or physical science. Modern computing environments are frequently composed of a range of different computers with different architectures. For various reasons, it is frequently desirable for an application (all or some parts) to be portable across different architectures. This can be done using the meta-data for a target platform

which can be used to select an architecture specific version of a task. In the proposed framework, this architectural type of behavioral meta-data analysis will be included.

b) Self-Adapting Numerical Software: Associated with the primary framework development will be work on a system called Self-Adapting Numerical Software (SANS). This work will explore additional uses of behavioral meta-data.

The concept of components includes an interface specification as part of its meta-data. We broaden this concept to include behavioral meta-data, which describes conditions for efficient use of a component. Unlike simple calling sequences, the behavioral meta-data interface specification can be expressed in non-programmatic terms; it can for instance refer to the numerics of the algorithm, or be derived from the application domain. This behavioral meta-data applies both to user data, as annotation of their properties, and to algorithms, as description of their behavior in the presence of certain properties of data. With meta-data attached to both data and algorithms, the programming environment can include a decision-making framework service task, called an Intelligent Agent, which picks that component out of a collection of candidates that is best suited to the user data.

While it is conceivable that component authors will supply such meta-data, asking users to annotate their data in this manner is unrealistic, and in fact we do not want to burden library writers with this task either. In Self-Adapting Numerical Software we let the Intelligent Agent learn from production runs, extracting information that over time tunes a number of heuristics.

IV. Discussion

A. Data management flexibility

One feature of object-oriented computing is that data is treated as a first-class citizen. However, the bundling of methods (user code) to a data set can be problematic. Data sometimes needs to be shared by user code written by different programmers. Replicated data sets also need to be shared on different contexts and platforms. On the other hand, the loose tie between data and code in procedural languages can lead to confusion and errors. The inclusion of a Data Set and a Task Programming Component was done to provide the advantages of both approaches. A Task does not need to own a Data Set. In this discussion, “own” is meant to infer that the task code requests that memory space be assigned for the data set, implicitly or explicitly. The inclusion of a Data Set Programming Component in a Context would make the framework responsible for memory allocation. The task would get the address of the data from a framework method. This allows multiple tasks that need to use the same data set to be included in the same context with minimum programming confusion.

B. Reference meta-data

The meta-data for Programming Components can include references to other Programming Components in the same and other families. This can be used to assist in application correctness. A Data Set can be configured for use with only certain Tasks. And vice versa, a Task can require the existence in any associated context of a specific Data Set. File, Message, Data Set Programming Components can all reference the same data format configuration meta-data. This will insure compatibility when moving data. If a Data Set and a File Programming Component reference the same data format, then a default I/O translation can be inferred and the user code does not include these format details.

C. Runtime efficiency

Just like the design of the code for a particular task algorithm is better if it is compatible with the architecture of the computer being used, the algorithm design of the composite application needs to match the communication resources of the heterogeneous, distributed environment being used. This can be a relatively simple procedure of comparing the data transfer rate requirement of a remote communication to the capability specified in the meta-data for a Network Programming Component.

But, within a local area network (LAN) the situation is not alway straight-forward. When file systems are shared, the choice of how to move files can be a problematic. For example, suppose a LAN contains 3 machines that use a shared file system with machine C being the file server. Suppose the user code on A writes a file and that file needs to be transferred to machine B so it can be read by another code. Since the file systems are shared, both copies of the file will reside on machine C—just in separate directories. Therefore, a local file copy (or a local file link) on machine C is the most efficient solution. However, many users will program the file transfer as an explicit remote copy which results in three network transfers—from A to B to C and back to A. Alternately, the programmer would need to program multiple solutions and the correct solution chosen depending on whether the file systems are shared or not.

This is one example where the proposed programming model will provide a benefit. A Site Programming Component can be defined that contains meta-data that describes the file system and computers within a LAN. The programmer will associate each Task Programming Component with a Context Programming Component which executes on a Platform Programming Component which is associated with the Site Programming Component. A File Programming Component can be associated with a Task or directly with a Platform. The end result is that the framework has enough information to take a generic file copy request from the programmer and generate the most efficient result.

The meta-data for the Platform and Site Programming Components can be generated by the system administrator for the LAN and shared by all applications. The application user would only have to provide the association of Context or File to the Platform. Even this may not be necessary if a virtual platform (which specifies requirements for a computer rather than a specific computer name or pool of computers) is defined for the application and the framework automatically matches it to a Platform defined to represent a physical computer. The generic file copy method would be implemented as part of

the framework. The result would be an efficient programming environment where the user can focus on the correctness of the algorithms and not the details of every distributed computing aspect.

This example shows that the benefits of separation of concerns are not limited to just organization of the application. Appropriate separation can make it easy and natural for different people to provide information for the composite application.

D. Resource management

One of the headaches of distributed computing is the linking of the computational needs of each context to a computing resource. Some users desire to request a specific computer, while others would like some system to select a machine by matching a specific requirement. The proposed model allows the Platform Programming Component to be configurable with one of several styles:

1) specification of a specific computer,
2) specification of a pool of computers, or
3) request the use of a resource scheduling system.

In addition, the linkages can be defined internally and externally to the user code. Also, the linkages can be statically or dynamically programmed.

This gives the programmer the ability to tightly integrate computer selection into a code or not. The meta-data for the Task or Context Programming Components can be used to define any specific computing requirements and the framework used to complete the resource selection for an application.

For example, a task that has been coded using a data-parallel, message passing system can require a computer with an appropriate architecture. The meta-data for a Task or an associated Data Set could include size information that could be used to determine which computing resource to select. Two tasks that need to pass data via message passing (or even files) could have a network requirement in their meta-data that could specify the locality (in terms of network performance) of appropriate computing resources. Clearly, the use of meta-data organized around appropriate programming entities will result in a programming environment where cleaner and more understandable task codes can be developed.

V. Conclusion

A. Summary

In this paper, it has been pointed out that a major problem with the development of many scientific applications is the need for a programming environment that supports development of code in an efficient manner. In particular, this means that the various aspects of the application are not so intertwined that the application becomes difficult to modify and otherwise manage.

A programming model that emphasizes separation of concerns has been proposed. The meta-data for the proposed Programming Components will be the nucleus of the approach to separate the details of the different coding aspects that make up an application. The Programming Component framework methods will support a variety of programming styles that allow complex composite applications to be built. This will include an approach where the task code can be developed with a minimal set of generic methods. This will allow very portable task code to be written.

The programming model is completed with the inclusion of behavioral meta-data and a system to use the analysis of behavioral meta-data. This functionality will support the use of tasks and data with which the programmer is not intimately familiar.

B. Implementation Plans

The planned framework will be build using standards being generated by two scientific community forums. The Common Component Architecture Forum is developing standards for the packaging of user code as components and frameworks that can efficiently execute those components. [16] The Global Grid Forum is developing standards to support distributed computing in heterogeneous environments that can include the use of multiple local area networks that are independently managed [17]. This means that security is an integral part of the design.

References

[1] F. Brooks, “No silver bullet: Essence and accidents of software engineering,” *Computer*, p. 12, April 1994.

[2] T. Sterling, P. Messina, and J. Pool, “Findings of the second pasadena workshop on system software and tools for high performance computing environments,” Center of Excellence in Space Data and Information Sciences, Goddard Space Flight Center, Greenbelt, MD, Tech. Rep. Report TR-95-162, 1995.

[3] A. Salas and J. Townsend, “Framework requirements for mdo application development,” in *7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, St. Louis, MO*, no. AIAA Paper 98-4740, September 1998.

[4] R. Weston, J. Townsend, T. Eidson, and R. Gates, “A distributed computing environment for multidisciplinary design,” in *5th AIAA/NASA/USAF/ISSMO Symposium on Multiple Disciplinary Analysis and Optimization, Panama City, Fl*, September 1994.

[5] G. Kiczales and e. a. J. Lamping, “Aspect-oriented programming,” in *Proceedings of the European Conference on Object-Oriented Programming (OOPSLA), Finland*. Springer-Verlag, June 1997.

[6] R. Filman and D. Friedman, “Aspect-oriented programming is quantification and obliviousness,” in *Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis*, October 2000.

[7] T. Eidson and G. Erlebacher, “Implementation of a fully-balanced periodic tridiagonal solver on a parallel distributed memory architecture,” *Concurrancy: Practice and Experience*, vol. 7, no. 4, June 1995.

[8] T. Eidson, “A programming environment for the development of large scientific systems on a distributed computing network,” NASA Langley Research Center, Hampton, VA, Tech. Rep. NASA SBIR 95 Phase 2 Final Report, Contract No. NAS1-97021 LaRC, March 1999.

[9] ——, “Implementation of wingbody/rlv application in lawe,” NASA Langley Research Center, Hampton, VA, Tech. Rep. Objective 2 Final Report, NASA LaRC PO: L10988, September 2000.

[10] J. Townsend, T. Eidson, and R. Weston, “A programming environment for distributed complex computing - an overview of the framework for interdisciplinary design optimization (fido) project,” NASA Langley Research Center, Hampton, VA, Tech. Rep. NASA TM 109058, December 1993.

[11] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[12] T. Eidson, “A component-based programming model for composite, distributed applications,” ICASE, NASA Langley Research Center, Hampton, VA, Tech. Rep. ICASE Report No. 2001-15, June 2001.

[13] R. Englander, *Developing Java Beans*. O’Reilly and Associates, Inc, 1997.

[14] J. Browne, K. M. S. Hyder, J. Dongarra, and P. Newton, “Visual programming and debugging for parallel computing,” *IEEE Parallel and Distributed Technology*, vol. 3, no. 1, 1995.
[15] C. Cicalese and S. Rotenstreich, “Behavioral specification of distributed software,” *Computer*, p. 46, July 1999.
[16] CCA, “Common Component Architecture Forum webpage,” in *http://www.cca-forum.org*, 2003.
[17] GGF, “Global Grid Forum webpage,” in *http://www.gridforum.org*, 2003.

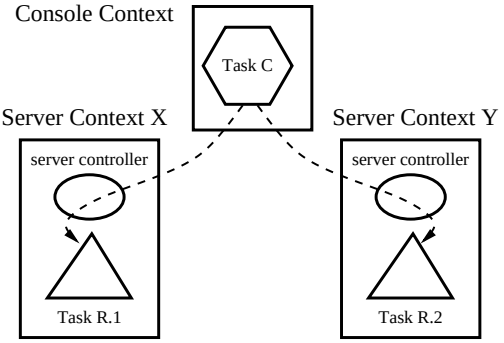

Fig. 1
TASK PROGRAMMING IN A FRAMEWORK

```
Arguments ri[2], ro[2]
Handle HX, HY, HR1, HR2

HX = Create_context("X")
   // HX - a handle for Context X
HY = Create_context("Y")
   // HY - a handle for Context Y
HR1 = Discover_task("R",HX)
   // HR1 - a handle for Task R.1
HR2 = Discover_task("R",HY)
   // HR2 - a handle for Task R.2

while(1) {
 Create_input(ri)
 Execute_task(HR1,ri[1])
   // Execute_task - "use" procedure
 Execute_task(HR2,ri[2])

 <other computations>

 Wait_on_elements(HR1, HR2)
  // Wait_on_elements - "wait" procedure
 ro[1] = Get_results(HR1)
 ro[2] = Get_results(HR2)
 if (Results_satisfactory(ro)) break
}
```

Fig. 2
PSEUDO-CODE FOR WORK-FLOW PROGRAM