Improvements in the Efficient Composition of Applications Built using a Component-based Programming Environment

Thomas Eidson, Victor Eijkhout, Jack Dongarra

I. Introduction

As computational science has evolved, applications are being developed that focus on more complex problems [1], [2]. This often results in the use of a set of element applications, that focus on a relatively narrow aspect, to build composite applications that are more complex in scope. This often requires for the various codes that make up the composite application to be executed on several computers located on a heterogeneous network of computers. These codes may be stand-alone executables or they may be a library of functions (or objects for object-oriented languages). These codes are often created by multiple developers, so efficient code sharing and good portability are often important requirements. The use of software component technology is viewed as one approach for developing a software development environment, or framework, that supports the development of such applications [3], [4], [5], [6].

Systems based on software components generally focus on the interface design of the user code packaged as a component based on some specification [7]. A framework can then be developed that supports the efficient integration of such components into a composite application. Often, a framework includes a registry service which catalogs available components along with appropriate information, or metadata, for use of those components. These systems can make the development of composite applications easier.

The software component model is sometimes viewed as providing a similar ease of use as that found with electronic components for audio systems and personal computers. This analogy has some merit, but there is a key difference. The functionality provided at each electronic component interface is limited to a few standards with new specification evolving slowly. In the computational science arena, especially at the research end of the spectrum, interfaces need to be very flexible as testing different ideas is often done. But even more important, the compatibility of a software component to a composite application can strongly depend on the internal behavior of that component. Determining if a software component is sufficiently compatible can require a great deal of testing and even modifying the composite application.

This research was supported in part by NSF NGS program under grant ACI-0203984 and in part by the AMS/MICS office of the DOE under contract DE-AC05-00OR22725 with UT-Battelle, LLC.

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

Our research [8] is focused on the development of a component-based framework that supports components with improved programming flexibility and with behavioral analysis capability. [9] In this paper, we will outline the approach that we are developing in these two areas. An important part of a programming system with such capabilities is the method by which information describing the programming characteristics of a software component is managed. This information is often packaged separately from the user code that expresses the primary algorithm of the component. It is referred to as metadata. In this paper, one focus will be on describing the metadata designs behind the prototypes that we are developing. Additionally, aspect-oriented programming (AOP) concepts will be used to motivate the design of a framework that supports better application modularity (or using an AOP term, better separation of concern) [10], [11].

II. Application Composition

One of the most important requirements of a well-designed application is that it be organized into appropriate modular units. Such a modular design is useful for a number of reasons, with the most important being that the code is typically easier for the developer and others to understand. But a modular design does not guarantee that the application is sufficiently easy and efficient to modify. As mentioned above, composite applications that are developed to study new phenomena need good modification flexibility. Developers will need to modify each code elements to determine the best algorithms. They might also need to combine code elements in a variety of different configurations. This means that the design of each element's behavior and its interfaces must support flexible composability requirements. Such applications may be used by groups and on a variety of platforms, so flexibility is also needed to support different user preferences.

Such flexibility starts with the design of the modular units. Most scientific applications are focused on computing algorithms that model some physical problem, and the code that directly implements those algorithms can be viewed as the primary aspect of the application. In the following discussion, this code will be referred to as the application 'tasks'. A well-designed task should typically compute a clearly defined segment of the overall application. Ideally, the task code would display a sense of completeness. Specifically, the algorithm being implemented would be relatively easy to discern from viewing the code. This would mean that the behavior of any

function calls should be able to be described in a succinct manner. However, modeling science can result in complex coding requirements where optimization and correctness demands result in code that is not so clearly designed. In some cases data can be very large, so a single copy must often be shared with different tasks in the application. This can result in compromises in coding style for some tasks that do not produce the optimum clarity. In other cases, the data has to be moved outside the scope of the task: written to a file, sent as a message, or passed as a function argument. For various reasons, the coding of such data transfers often results in intertwined code. This intertwined code can result in a loss of clarity in understanding the primary aspect of the task and make it more difficult to make code changes. This data transfer code can be though of as one type of secondary aspect of the user code.

The result is that a great deal of scientific applications are difficult to port to other machines, share with other users, and integrate with other applications. The developers want to focus on developing the best tasks (where they have expert skills) and not be distracted with the design of the secondary tasks. While object-oriented programming has provided some constructs which help, much more needs to be done. The research being discussed herein is focused on developing programming techniques that promote the development of applications with good portability characteristics which can be more easily shared with others and integrated into composite applications. The emphasis will be on supporting a programming style where the primary tasks and the secondary aspects can be managed separately.

A. Distributed Computing Aspects

It can be a subjective decision as to what code is characterized as part of the primary tasks and what code is associated with secondary aspects. With this caveat, one example of a secondary aspect is the modification of a straight-forward implementation to give better performance for a particular compiler or hardware architecture. Another example is error handling code that allows an application to close gracefully rather than crashing when an error occurs. However, a large group of aspects deal with communication of data between different tasks as discussed in the previous section. Programs written for distributed environments usually include a large number of these type of aspects. The approach discussed below will focus on these data communication aspects.

As computer hardware technology evolved, distributed computing became an alluring option. At the expensive end, special purpose machines need to be shared by users located at different sites. At the less expensive end, there is an increasing number of underutilized machines with a useful performance level. However, to be useful, it is usually necessary that the application be distributed over a number of machines, and distributed applications can be difficult to develop and manage. Distributed applications involve more resources than a single machine application and the result is more secondary aspects which are required to manage these resources. For a single machine application, the main secondary aspect is managing input/output to files. For a distributed application, the application not only needs to use files but it needs to know where the tasks are that use them, how those tasks are grouped into executables, where those executables are located, and other information not relevant to the single machine environment. Multiple copies of data and code are often needed. Error handling and debugging are more difficult.

B. DIST Programming Model

The DIST Programming Model was a first attempt at managing these distributed computing aspects [12], [13]. The DIST Programming Model evolved as part of the development of a general programming framework for the development of large, distributed, composite applications. The primary user code was encapsulated as a Task Programming Component. The other coding aspects were encapsulated into other Programming Components.

1) A *Task Programming Component* maps to the basic task concept described above. It is the primary abstraction that encapsulates user-written code. The user code could be single-threaded, multi-threaded, or data-parallel. A popular approach to designing programming systems is package task code as a software component.
2) A *Context Programming Component* is defined as a collection of tasks and data packaged for execution and interaction with other tasks. A Unix process is an example. It is often convenient to design a context as a server that can dynamically load tasks and create memory to store data values.
3) A *Memory Programming Component* is used to manage user defined datasets that need to be shared with other tasks in the same context. By managing the primary copy of shared data outside the scope of a task, each task gains portability. This eliminates ownership of the data as a composability problem. Clearly, appropriate synchronization must be part of the design. Both copy and direct access (via pointers) techniques can be made available to support parallel usage and performance requirements.
4) A *Platform Programming Component* encapsulates one or more computers managed as a single entity that is connected via a network to other platforms. A Platform is managed as part of a site (essentially, a local area network with some shared file systems). In addition to describing the details of a physical or virtual computer, the Platform includes information about file systems, resource schedulers, and potentially other site resources.
5) An *External Programming Component* encapsulates software systems and hardware devices outside the scope of a context where data can be sent to or received from. The primary example is a file.
6) An *Event Programming Component* is used to defined synchronization signals that are shared by different tasks.

Each Programming Component type encapsulates a physical or conceptual entity that needs explicit management in a distributed application. Associated with each Programming

Component type are a set of methods to manage that entity and a set of metadata to allow one to configure different entities of that same type. The entities can be specific or virtual. For example, a specific Platform definition would describe a specific computer and specify its network address. A virtual definition would give requirements that a scheduling system could satisfy at runtime. Framework methods are designed to provide the basic distributed computing requirements. For a Context instance, this includes starting and stopping the associated executable code. For a Task instance, this includes starting and stopping user methods or functions along with moving input and output arguments between the Contexts where the calling and the called code reside. For a Memory instance, methods are needed to move data between a Memory instance and a Task instance, as well as between two Memory instance which might be in different Contexts.

For each Programming Component type, one or more definitions can be defined via metadata and named. The names of these Programming Component definitions are then used with framework methods in user code to identify each particular definition. In the user code, an instance of a particular Programming Component definition can be created and named. Framework methods associated with that Programming Component type can then be executed on that instance object. A framework compatible with the DIST Programming Model will execute these methods and provide any underlying distributed computing functionality. Because of the choice of Programming Component types and the associated metadata, a framework can be designed to provide this distributed computing functionality with a good degree of efficiency.

III. Applying Aspect-Oriented Programming concepts

A. Aspect-Oriented Programming

The DIST Programming Model supports a programming style with good composability and portability characteristics. An application that uses DIST has much of the distributed programming detail removed from internal locations in the task code and located in metadata files where it can be more easily managed. However, specific Programming Component methods still were dispersed in the user code and portability still needed improvement.

For example, a task might be developed to use an External Programming Component to write data to a file. When shared in another application, this task might be required to send the data as a message to another task. The task would then need to be modified. It is such a modification that generates the concern. Often when such modifications are made, the primary aspects of the tasks are accidentally altered. The debugging process then needs to be repeated and this can be very expensive.

As mentioned above, it would be much better if the primary and secondary aspects could be managed separately. For code correctness and performance reasons, this can be difficult. Therefore, the different aspects are often programmed in a cross-cutting style. Managing these cross-cutting concerns is just one example of the programming problems that are being addressed by a area of research which is referred to as Aspect-Oriented Programming (AOP) [10], [11]. An AOP-based system would manage the task and aspect code separately to support efficient code development and modification. A programming and execution framework would support the interweaving of the task and aspect code so that correct and efficient execution would result. This interweaving could occur during compilation or at runtime or both.

B. The Nautilus Framework

In the Nautilus project, AOP concepts are being used to extend the DIST Programming Model. The focus is on replacing the use of Programming Component methods that are specific to each Programming Component type with generic methods. When the methods for the various Programming Component types are analyzed, it can be observed that they all involve importing or exporting data and defining functionality requests that need to be executed outside the scope of that task. The portability and flexibility concerns, mentioned above, are directly related to how much of the detail of each external request is hard-coded inside the task.

A generic method approach is being incorporated into the DIST model to minimize those details. Many import/export operations involve moving data. At the point where the request is passed to the execution system, the data is passed by specifying it in terms of task variables which take the form of pointers or memory addresses. [This is even true when data is passed-by-value. The execution system just uses the memory addresses to make a copy of the data in that case.] An alternate approach would be to organize any data that needs to be passed out of the scope of the task into datasets. Each dataset would consist of a set of elements. Each element would consist of an array of variables of a single base data type along with a label (i.e, a string variable). The task would create such datasets, name them, and expose their memory addresses to the framework as part of a registration process. This would be allow data references to be passed to any execution system or framework as labels rather than memory addresses when done as part of a request for external functionality. This supports a distributed execution environment since labels can be understood by executing code in different memory address spaces (i.e., different processes). It also supports the use of AOP concepts. Aspect code can be written that executes secondary aspects outside the scope of the primary task. The framework would intertwine the execution of the aspect and the primary task code in an appropriate manner. The aspect code would just execute on data provided by the task at some appropriate point. The use of labels to specify the desired data to be used by the aspect allows for a more flexible system to be designed.

The user task needs to define and to communicate these appropriate points to the framework. In the AOP literature, such points are referred to as join points. In Nautilus, they are just framework methods that passes a unique label to the framework to identify a location in the user algorithm. Separate aspect code can then be generated that maps a desired operation and any associated dataset labels to each

join point label. An AOP implementation would vary from such a simple mapping to a more complex approach where conditional statements are included.

A set of join point and dataset definition methods could be enough. But, this would require that all data access synchronization be specified as part of the definition maps for a join point. By adding lock and unlock functionality to the dataset design, it is anticipated that a more flexible programming system will result. Both approaches will be tried in the Nautilus prototype being built.

The implementation will initially take two forms. The base form will be a set of framework aspect methods that can be configured by the application developer using metadata. This will provide the simple mapping functionality described above. The programmer will provide a list of mapping instructions that specify the name of methods to execute at each join point along with the name of the datasets to use with those methods. For more complex functionality, such as conditional execution of an aspect at a join point, an aspect method library will be developed. This library will allow the user to write an aspect code that would typically run in parallel with the primary task. For example, this user aspect code would interject appropriate operations at each join point in the task code.

C. Discussion

Historically, all information requests needed to augment the primary computations of a user task have been done with some type of procedure call. In most languages, these procedure calls and any associated data were passed to an executing framework (the operating system in most cases) as some type of pointer. This has proved to be a very efficient programming and execution approach in single machine environments. But for distribute environments, pointers generally cannot be passed from one executable to another. Essentially, the request on one machine needs to be put in a form that can be shipped to the target machine and then translated back to form where the code on that target machine can execute the desired procedure. One approach is to use an interface definition language(IDL) and define any needed functions or methods in this 'super-language' so that an IDL compiler can generate appropriate code for used in the calling and the called executable [5].

The approach described below supports applications that need a solution that a more flexible approach to code integration. Consider the follow form:

```
<output dataset label> =
  framework_method (<function label>,
                    <input dataset label>)
```

The 'framework method' represents a small set of traditional style procedure calls that could be used to request any functionality that is executed outside the scope of a task. The specific functionality could be expressed by the <function name> argument. It would just be a portable label (i.e, a string variable). Just as was the case for data describe above, the use of a label rather a function pointer has advantages for implementing framework that support distributed computing and aspect-oriented programming. The use of separate datasets for input and output arguments makes it clear what data needs to be moved.

For applications that include a large number of function calls with a variety of different arguments, one might think that this would result in maintaining a large number of datasets. One might need to create or modify a dataset for each remote request. However, the framework could be designed to support argument formats. Using argument formats a super dataset could be programmed that contained the input arguments (and even the output arguments) for a set of functions. The metadata for each function could include the name of an argument format. The argument format would specify those element labels needed for the input and output arguments of that function. Aspect code could be used to select different element labels for the same function at different join points. The user task could be written in a relatively generally style. External metadata could then provide the specificity to complete the application. The advantage is that a task can be more easily modified to test different options or to be integrated in other applications.

The design being developed in the Nautilus framework is exploring many of the possibilities suggested above. The target 'framework methods' will include the join point concept along with support for dataset management and usage. The design will include all the original DIST system Programming Components and framework methods. But, the implementation being built will support the execution of these methods as aspect code triggered by execution of a join point located in the task code. In many cases, the distributed computing functionality can be requested by creating a simple metadata file. For example, a metadata file could contain a list of Contexts, each with an associated Platform. This list would specify a set of executables along with the machine on which to start that executable or a procedure to determine that machine. The metadata could also specify a set of Tasks to pre-load (so that the executable defined by a Context could dynamically link to a object library). Other metadata information could specify that files be pre-staged to designated Platforms (actually to a file system associated with the machine represented by that Platform). Such aspect metadata could be used to initialized the remote executables of a distributed application. Alternately, it could be triggered by a single execution of a join point to start additional resources. This is just one example of where the extension of the DIST model using AOP concepts can generate an application with significant programming flexibility.

Ultimately, it is believed that the proposed approach will result in a programming style that supports a high degree of programming clarity. When a developer writes a task, they will focus on the coding of the primary algorithm along with defining datasets and join points to support any potential sharing of data with another task. This is similar to what a user does when a function call is inserted into the task. The difference will be that the user will focus more on data exchanges and the timing of those exchanges rather than the behavior of how that data is created or used outside the scope of the task being created at that moment. That data exchange

may be done by the execution of a local or remote task at runtime, but it could also be done by reading or writing a file (or some equivalent entity). This decision of how the data will be exchanged can more easily be delayed because of the programming flexibility that results from an AOP-based framework design. This will be particularly important when the choice of the specific remote task requires a behavioral analysis technique were data needs to be collected by running several candidate tasks in the target application to see which is more suitable. This is the type of analysis discussed in the Behavioural metadata section below.

The above approach should be viewed as an experiment in new programming techniques rather than as a prototype for a new language. The basic goal of an AOP system, a better modular application design, is clearly desirable. The ultimate value of AOP will depend on whether practical approaches can be implemented. While programming each aspect of a system separately has conceptual merit, it can also lead to disaster. Most experienced programming are aware of many cases where a high-level programming choice did not give the expected result when the compiler translated that high-level instructions and integrated them with the other parts of the application.

For this reason, the approach taken is not to develop just a set of high-level instructions that implement aspect weaving in a black-box mode. For a few simple cases, framework methods will provide aspect weaving capability. For the more complex cases, the emphasis is to implement support for the programmer to develop their own aspect weaving code. Once the value and practicality of aspect weaving is learned, more sophisticated language approaches can be implemented.

Clearly, there will be some negatives to this approach. As the initial implementations will all be done using runtime libraries, execution efficiency may suffer. However, the potential exists for the some of proposed techniques (along with ideas from other AOP research) to be integrated into compilation systems. Ideally, new languages could be developed that better support distributed computing and aspect-oriented programming.

IV. Behavioral metadata

Our proposed programming framework can also govern the insertion of numerical capabilities into application components. This process is not straightforward, since often there are several numerical algorithms that will realize the same input-output specification, but do so with radically different degrees of efficiency. Deciding on the right algorithm is therefore of tremendous importance. Unfortunately, the optimal choice is inherently intractable, depending in complicated ways on the problem data, and in practice we can only hope to make a good enough choice. Even this more modest aim is one that is difficult to automate. For small-scale projects the proper choice of numerics involves extensive experimentation, and for larger-scale projects often a numericist is included on the payroll to arrive at an efficiently executing application.

Recent experiments have shown that this degree of human intervention can be considerably lessened, by employing non-numerical data analysis techniques in the field of solving systems of linear equations by iterative methods. Work by Bhowmick *et al* [14], [15] uses statistical techniques to construct multi-methods, ranking iterative methods in some order of reliability, an idea that was used in the LinSol package [16] but only based on heuristic reasoning there. A more dynamic approach, basing the ranking of methods on the properties of the input data, was taken by Xu *et al* [17] and Eijkhout and Fuentes [18]. Experiments by Eijkhout and Fuentes have shown this approach of applying non-numerical techniques to numerical decision making to be especially promising.

Our development of behavioral metadata in the programming framework is intended to support such approaches.

A. Components and interfaces in numerical decision making

In the traditional way of calling numerical libraries straight from the application code, the only defined interface is that of the problem data. Typically this is stored in some sparse matrix format and using arrays for any vectors passed. However, implicitly there is at least one more interface. Many numerical routines take parameters that control their inner workings, such as the restart parameter in GMRES, the ordering of a direct LU factorization, or the number of fill-in levels in an ILU preconditioner. Absent any decision-making support, such settings are supplied by the programmer. In our notion of a framework we want to move to a scenario where these settings are programmatically supplied. A component supplying these parameters could additionally make the decision which of a number of available algorithms to instantiate in the context of the current problem.

This component, which we will call an ‘(Intelligent) Switch’, has two interfaces, one on the side of the numerics, and one on the application side. For now we will largely be concerned with the latter; we will address the former in future research.

The Intelligent Switch is concerned with dynamic numerical decision making. For this, the application-side interface accepts minimally the problem data. This, however would put the burden of the problem analysis on the Switch, which is not the appropriate place for it. Rather, we posit the existence of Analysis Modules which accept the problem data (or subsets thereof) and return metadata: higher level descriptions of the problem data. Since this kind of metadata is of a higher level than strictly describing the data, and is intended to influence the numerics of the application, we call this ‘behavioral metadata’.

B. Behavioral metadata

Behavioral metadata is different in nature from the metadata described in the earlier part of this paper, and from the usual notion of metadata as giving a full description of problem data. We envision two kinds of behavioral metadata:

1) Information that is derived from the problem data, such as matrix norms or estimates of the operator spectrum, and
2) Information that is known to the generating application, but which typically does not get included in the problem

data, such as the nature of the discretization scheme used.

We will devote a few words to both categories.

1) Derived metadata: There is a lot of information about problem data that could conceivably be useful to numerical software, such as information about the structure of the problem, or spectral information about the operator. (See section IV-C for discussion of our initial core categories of derived metadata.) Some of this information, such as structural properties, can be derived from a matrix in time proportional to the number of nonzero elements. Spectral data, on the other hand, can not be found exactly in a time essentially less than that required to solve the problem, so we have to resort to heuristic estimates. In between these two categories lie such data as the norm of the symmetric part of an operator, which is moderately tricky to compute, especially in a parallel context.

Generation of derived metadata requires only the problem data as input to the analysis modules; the metadata is then the formal description of information passed from the analysis modules to the switch component.

2) Application metadata: The second kind of behavioral metadata does not consist of derivable data, but rather of information that is normally lost in the interface between the physics and numerics component. Typically examples are the coordinates of grid points, which can be used for geometric domain decomposition, or the nature of the finite element discretization, which information can be used by multigrid codes, or incomplete factorization routines.

In this case of application metadata, the analysis modules do not derive the metadata, but rather perform at most a translation. Both interfaces of the analysis module now use behavioral metadata, though probably of different categories on input from on output. (See below for a further explanation of metadata categories.)

C. Core metadata categories

In [18] we described a storage format and API for dealing with behavioral metadata. We also proposed a set of core categories of metadata that we think will be useful in many numerical linear algebra contexts. Here we briefly reiterate these categories.

- In case we are dealing with problem data that is stored on file, rather than in data structures, we can give information about the file, such as to note if elements are sorted by row or column.
- Often, in such contexts as nonlinear solves or time-stepping methods, we are dealing with a family of matrices that are based on the same sparsity structure. Thus, we propose a category of metadata to deal purely with structural issues.
- Next we distinguish between information that can be computed exactly, and information that can only be estimated. We posit a category of numerical metadata (that is, dependent on actual values of matrix elements, rather than just structure) that can be computed exactly, and in a low time order, typically of the order of the number of nonzero elements. This category contains such information as matrix norms (although it should be noted that the 2-norm and spectral radius are in general not easy to compute) and amount of diagonal dominance. Matters of symmetry, both noting whether a matrix is symmetric and giving the norm of the symmetric and anti-symmetric part, also belong in this category.
- Finally, we propose a category for spectral matrix data, giving estimates for such quantities as the location of the ellipse enclosing the spectrum, and the departure from normality. Since the estimates given for these quantities depend on the algorithm used to derive them, our metadata format has provisions for annotating the value with the ‘signing authority’; see [18] for further details.

V. Summary

In this paper, the efficient development of composite applications has been discussed with the focus being on the use of software components to package the user code that is used to build the composite applications. Because of the complex nature of scientific applications, using software components in a plug-and-play style is not so simple. The design of the software components (or user tasks) and the frameworks used to integrate them into a composite application need to support a great deal of flexibility. This is needed to allow efficient experimentation in the development of the application. Additionally, frameworks need to assist the application developer beyond just ensuring interface compatibility. Matching the internal behavior of two user tasks that are coupled in a composite application is equally important. While a completely automatic behavioral analysis may never be a reality, frameworks can include sub-systems that provide the user with insight as to the compatibility of the integration of a task into a composite application.

The goal of the research program described in this paper is to address the above issues by applying several relatively new techniques to explore the design of components, frameworks, and the associated programming styles. Metadata is being used extensively to describe various elements of an application and a framework and software tools are being developed that use this metadata to augment the execution of the user tasks. Aspect-oriented programming concepts are being used to support better modularity and portability in the design of components and composite applications. Behavioral analysis techniques are being used to further enhance portability by including knowledge about the use of an task in its metadata.

References

[1] 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.

[2] J. Stewart and H. Edwards, “The SIERRA framework for developing advanced parallel mechanics applications,” in *Proceedings of First Sandia Workshop on Large-Scale PDE-Constrained Optimization*. Springer Lecture Notes in Computational Science and Engineering, 2001.

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

[4] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, , and B. Smolenski, “Towards a common component architecture for high-performance scientific computing,” in *Eighth IEEE International Symposium on High Performance Distributed Computing*, August 1999.

[5] J. Siegel, *CORBA: Fundamentals and Programming*. John Wiley and Sons, 1996.

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

[7] "Common Component Architecture Forum webpage," in *http://www.cca-forum.org*, 2003.

[8] T. Eidson, J. Dongarra, and V. Eijkhout, "Applying aspect-orient programming concepts to a component-based programming model," in *Proceedings of 17th International Parallel & Distributed Processing Symposium*, April 2003.

[9] C. Cicalese and S. Rotenstreich, "Behavioral specification of distributed software," *Computer*, p. 46, July 1999.

[10] G. Kiczales and J. Lamping et. al., "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming (OOPSLA), Finland*. Springer-Verlag, June 1997.

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

[12] 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.

[13] ——, "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.

[14] S. Bhowmick, P. Raghavan, and K. Teranishi, "A combinatorial scheme for developing efficient composite solvers," in *Lecture Notes in Computer Science, Eds. P. M. A. Sloot, C.J. K. Tan, J. J. Dongarra, A. G. Hoekstra, Number 2330*. Springer Verlag, Computational Science ICCS 2002, 2002, pp. 325–334.

[15] S. Bhowmick, L. McInnes, B. Norris, and P. Raghavan, "The role of multi-method linear solvers in pde-based simulations," in *Proceedings of the 2003 International Conference on Computational Science and its Applications, ICCSA 2003, Montreal, Canada May 18 - May 21, 2003, Lecture notes in Computer Science 2677, Editors V. Kumar, M. L. Gavrilova C. J. K. Tan, and P. L'Ecuyer*, 2003, pp. 828–839.

[16] H. Häfner, W. Schönauer, and R. Weiss, "The portable and parallel linear solver package LINSOL," in *Proceedings of the 4th European SGI/Cray MPP Workshop, IPP R/46, Oct. 1998*, pp. 242–251.

[17] S. Xu, E.-J. Lee, and J. Zhang, "An interim analysis report on preconditioners and matrices," University of Kentucky, Lexington; Department of Computer Science, Tech. Rep. 388-03, 2003.

[18] V. Eijkhout and E. Fuentes, "A proposed standard for numerical metadata," Innovative Computing Laboratory, University of Tennessee, Tech. Rep. ICL-UT-03-02, 2003.