A Case Study in Application Family Development by Automated Component Composition: h-p Adaptive Finite Element Codes

Nasim Mahmood[1], Yusheng Feng[2], and James C. Browne[1]

[1] Department of Computer Sciences,
University of Texas at Austin, Austin, Texas 78712
{nmtanim, browne}@cs.utexas.edu
[2] Institute for Computational Engineering and Sciences,
University of Texas at Austin, Austin, Texas 78712
feng@ices.utexas.edu

Abstract. This paper reports a case study in automated composition of application families from components. The case study composes multiple instances of an h-p adaptive finite element code. An application family is represented as a structure of components. Each component is encapsulated with an interface giving a semantic specification of the properties and behavior of the component. Instances of the application family can be automatically assembled from a library of components by a compiler and the application instance can be optimized by component replacement during runtime through runtime component selection and binding. The case study demonstrates the benefits of the component composition approach to application family development and shows that execution efficiency is maintained or improved by the componentized development process.

1 Introduction

1.1 Applications as Families of Programs: Problem, Approach and Case Study

Application packages such as h-p adaptive finite element codes can be applied to a wide spectrum of problems in engineering and sciences [1,2]. These packages are usually composed of a large number of parameterized functions including mesh generation, element matrix generation strategy for h-p adaptation, methods for solution of the resulting linear system of equations, etc. These application codes must operate robustly and efficiently on a wide spectrum of problems and on a wide spectrum of execution environments. The properties and behavior of the program may vary widely with the mesh, with the model/equation set, the properties of the material or materials composing the system which is the subject of the computational model, etc.

The current practice in development of h-p adaptive codes is to construct them as an integrated and comprehensive package of functional modules based on common, shared data structures. A package which is robust and offers a wide spectrum of implementations for each of the possible functions may be very complex and very difficult to debug and to maintain and modify and these codes are often sub-optimally efficient on many of the problems to which they are applied and many of the execution environments upon which they may be hosted.

We address these problems through automated component-oriented development. We demonstrate that such automation is feasible, and execution efficiency is maintained or improved by the componentized development process. The semantic parameters of choice in our case study include: model or equation set, problem geometry, mesh structure including mesh generation, strategy for h-p adaptation and methods for solution of resulting linear system of equations. Each functionality is represented as a separate, semantically parameterized, component. There may be several implementations of each component, for example, there may be several different solvers (direct or iterative).

Each component is encapsulated with an interface which specifies its properties. This encapsulation is based on a domain analysis of the problem set the application is intended to address and a domain analysis of applicable computational methods. An application family is developed by specifying properties for each component and invoking a compiler which automatically selects appropriate components through the use of information on the components specified in the interfaces of these components. The compiler can generate a serial or parallel (either multithreaded or MPI) code as desired. The codes may be adapted to evolution of the meshes and approximation functions by monitoring of the code and runtime replacement of components in the computational model.

The approach is illustrated by a case study of its application to a simple but representative h-p adaptive code which contains all of the functions typically found in such codes. The code is reverse engineered to extract components which can be characterized in terms of the domain analysis. Several versions of the application family are realized through the compilation process and the properties and performance of the instances are presented. It is demonstrated that application instances which are efficient for multiple cases of the reachable application family are readily generated.

2 Approach: Component-Oriented Development with Semantic Interfaces

This section sketches the concepts of component-oriented development which form the basis of this case study and describes the implementation technology. A more complete specification can be found in [7].

2.1 The Interface Definition Language

The concepts of the interface definition language which specifies the semantic properties of the components are sketched in the following.

Component: A component is one or more sequential computations, an interface which specifies the information used for selection and matching of components and a state machine which manages the interface, the interactions with other peers and the invocation of the sequential computations.

Associative Interface: An associative interface [7] encapsulates a component. It describes the behavior and functionality of a component. An associative interface consists of an **accepts** specification and a **requires** specification.

Accepts Specification: An accepts interface specifies the set of interactions in which a component is willing to participate. The accepts interface for a component is a set of three-tuples (profile, transaction, protocol). A profile is a set of attribute/value pairs. A transaction is a somewhat extended procedure call and a protocol is a state machine defining a sequence of interactions.

Requires Specification: A requires interface specifies the set of interactions which a component must initiate if it is to complete the interactions it has agreed to accept. The requires interface is a set of three-tuples (selector, transaction, protocol). A selector is a conditional expression over the attributes of all the components in the domain.

Properties of implementations such as degree of parallelism for a given component are also specified in the associative interface as runtime determined parameters.

2.2 Compilation Process

Each attribute name in the selector expression of a component behaves as a variable. The attribute variables in a selector are instantiated with the values defined in the profile of another component. The profile and the selector are said to match when the instantiated conditional expression evaluates to true.

The source program for the compilation process is a component which implements initialization for the program and has a requires interface which specifies the components implementing the first steps of the computation and one or more libraries to search for components. The target language for the compilation process is a generalized data flow graph as defined in [9]. The generated data flow graph is then compiled to a parallel program for a specific architecture by compilation processes implemented in the CODE [9] parallel programming system.

2.3 Runtime Adaptation

The component-composition approach to application family development enables substitution of components implementing different algorithms during execution.

Most operating systems enable runtime linking of components to executable images. The requirement is to identify components which need to be replaced and to specify the properties of the component which is to be substituted for an existing component. Additionally the programs must be instrumented to acquire and analyze the execution properties and behavior of the components of the system.

Composition of a program from components enables and facilitates each of these tasks. Monitoring can be done on a component by component basis; components whose behavior is unlikely to vary need not be monitored. The monitoring code is readily generated by the compiler on a component by component basis. The required analysis and actions can be provided in a separate component or components.

3 Case Study

3.1 Description of the h-p Adaptive Finite Element Code

This case study in application family development is based on an h-p adaptive finite element code structure developed in [3,4,5]. These packages have a common data structure in one-, two-, and three-dimensional space. The major logical components include mesh generation, problem definition, shape function definition, and element routine, linear system of equation solver, error estimation module, and h-p adaptation module. We have used the one-dimensional code in this case study since it has the same structure as the two-D and three-D codes but is of considerably smaller size.

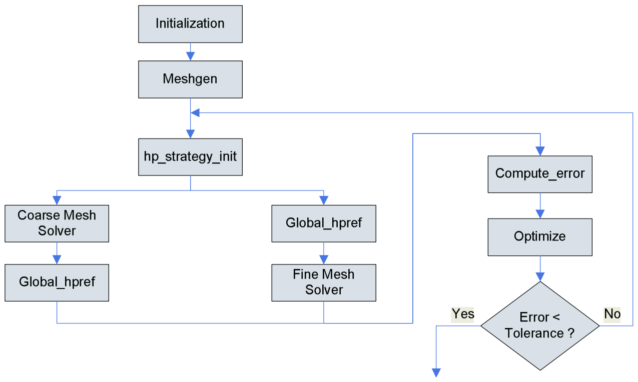

Fig. 1. Workflow Diagram for h-p Adaptive Finite Element Code

3.2 Componentization of the h-p Adaptive Finite Element Code

The set of components is determined by constructing a workflow diagram for the application in which each logical function is identified as a component. Figure 1 is a workflow diagram for a family of codes implementing *h-p* adaptive codes. Figure 1 and the components in Figure 1 were obtained by reverse engineering the one-dimensional code described in the previous section. This componentization does not represent the finest granularity of functional decomposition. The "Coarse Mesh Solver" and the "Fine Mesh Solver" each contain three logical functions, the computational model, the element generator for the stiffness matrix and the solver for the stiffness matrix. Componentization was stopped at the level shown in Figure 1 because component extraction by reverse engineering of the existing code was laborious and because this componentization enables practical experiments in componentization.

3.3 Experiments

The experiments illustrate composition of programs implementing a sequence of models, compile time choice of linear solvers and runtime substitution of the linear solver.

Compile time selection of linear solvers is illustrated by composing application instances first using a direct solver for the coarse mesh and a conjugate gradient solver with a diagonal preconditioner for the fine mesh. Runtime replacement (and optimization) is illustrated by replacement of the direct solver by the conjugate gradient solver after the first cycle of the adaptation demonstrates that the direct solver is not an efficient choice.

Composition of applications based on different computational models for a physical system is illustrated by composing a sequence of applications using successively more accurate models for bioheat transfer. We consider a set of bioheat transfer equations ranging from simple conductivity (Poisson) to incorporation of blood perfusion (Pennes Equation) to incorporation of artery-vein countercurrent (Weinbaum-Jiji Equation [5]).

These models represent progression of complexity and accuracy from the simple Poisson model through the Pennes and Wein-Jiji models. The experiment compares a standard metric resulting from solution of each of the models.

3.4 Illustrations of Automated Composition

3.4.1 Compile Time Selection of Solver and Model

The component library is initialized with two solvers: i) A direct solver that uses LU factorization and back substitution and ii) A Preconditioned Conjugate Gradient (PCG) solver that uses a diagonal preconditioner. Each of the four models sketched in Section 3.3: i) Laplace model ii) Poisson model iii) Pennes model and iv) Weinbaum-Jiji model have been incorporated into a component. The componentization of the h-p adaptive code leaves the model and the solver in the same component although they could readily be separated and would be separated for a production implementation. There are therefore eight implementations of the solver component. Each can be used for the coarse or fine solver so long as the model is the same for both the coarse and fine meshes. These eight implementations were encapsulated using the interface definition language of PCOM^2. A component that needs a particular combination of solver and model expresses that requirement using the selector interface. The selector of a component that requires a direct solver and Poisson model is shown below (only the attributes part is shown for space limitation).

```
Selector:
    string domain == "application";
    string component == "solver";
    string solver_type == "Direct"
    string model == "Poisson"
```

Similarly a PCG implementation of a solver that uses a Laplace model expresses that information in the profile of that implementation.

```
Profile:
    string domain = "application";
    string component = "solver";
    string solver_type = "PCG"
    string model = "Laplace"
```

The compiler chooses the appropriate component as described in Section 2.2. By changing the selector section of a component the appropriate implementation can be chosen at compile time.

Table 1 compares the solutions obtained from application family instances based on each of Poisson, Pennes and Weinbaum-Jiji computational models. Using Weinbaum-Jiji as a base model, we compared the solution in $H^1(D)$-norm. Table 1 indicates that differences are significant. These quantities in percentage can be used as a criterion for the decision-making in model selection. For example, if the acceptance criterion is set to 20%, then we need to reject both Poisson and Pennes models with respect to more accurate Weinbaum-Jiji model.

Table 1. Properties of Solutions from Multiple models

Model	Poisson	Pennes	Weinaum-Jiji
Solution Norm	0.18787E+06	0.18348E+06	0.14895E+06
Percentage	26%	23%	-

3.4.2 Runtime Optimization by Component Replacement

The PCOM^2 compiler automatically generates performance measures for the execution behavior of each component. This information can be used to determine whether a currently loaded component is performing efficiently and/or robustly. When it is determined that a change of algorithm is needed, the dynamic loading capability of the PCOM^2 runtime system can be used to dynamically replace an implementation of a component. The implementation of the solver component incorporated code to load libraries at runtime depending upon argument values in the transaction specification. Based on the argument (a domain attribute) the implementation can either run the direct solver or load a PCG solver from the library and invoke it. Similarly the PCG solver can be directed to replace itself by a direct solver. In the illustration reported here, during the first iteration the coarse mesh was solved using a direct solver and the fine mesh was solved using a PCG solver. But for large mesh sizes the direct solver component may take a longer time to solver the coarse mesh than the PCG solver takes to solve the fine mesh. After the first iteration, the runtime of the direct solve of the coarse mesh and the PCG solve of the fine mesh are compared component are compared in the *optimize* component, “optimize.” If it turns out that the direct solve of the coarse mesh is too slow, an appropriate argument is passed to the coarse mesh solver so that it can load the PCG solver using dynamic loading from the library on the next mesh refinement iteration.

Table 2. Execution Time Improvement with Dynamic Solver Replacement

Iteration	Coarse Mesh Solve	Fine Mesh Solve	Total Solve Time
1	2401x2401Direct 3.162 sec.	5401x5401PCG 1.199 sec.	4.361sec.
2	2404x2404PCG 0.536 sec.	5404x5404PCG 0.972 sec.	1.508 sec.

Table 2 summarizes the results of some experiments with dynamic solver replacement. An appropriate choice of solver cuts the time for solution down by nearly a factor of three.

4 Related Research

For a survey of the research on component-oriented development focusing on generation of parallel and distributed programs, please see [7]

There has been very little research utilizing dynamic replacement of components to enhance performance or robustness of scientific and engineering applications. Adve's PCL [8] system enables runtime adaptation of task graphs at a finer level of granularity (the basic block level).

AspectIX [10] offers the ability to replace an implementation at runtime. The functional and configuration interface in AspectIX is similar to the transaction and attributes of the profile in PCOM^2. The transaction provides the syntax of a component and the attributes expresses the semantics in the program domain. AspectIX uses the interface information at runtime. P-COM^2 integrates runtime and compile time composition.

5 Conclusions

The feasibility and the potential benefits of automated compiler-based composition of instances of application systems from libraries of components has been demonstrated. The concepts and their applications are quite simple and readily accessible to application developers. Modification and maintenance of families of application systems is simplified. Runtime adaptation at the component level is shown to be straightforwardly implemented and to offer the potential of substantial performance benefits.

Acknowledgements

This work was accomplished through support from ITR NSF grant number 0205181 , "A Computational Infrastructure for Reliable Computer Simulations." NGS NSF grant number 0103725 "Performance-Driven Adaptive Software Design and Control." Additional support came from the Defense Advanced Research Project Agency (DARPA) under Contract NBCH30390004.

References

1. M. Ainsworth and J.T. Oden, A Posteriori Error Estimation in Finite Element Analysis. John Wiley & Sons, New York, (2000).
2. I. Babuska, and T. Strouboulis, Finite Element Method and its Reliability. Oxford Univ. Press (2001)

3. L. Demkowicz and C. W. Kim, 1D hp-Adaptive Finite Element Package. Fortran 90 Implementation (1Dhp90), TICAM Report 99-38, The University of Texas at Austin (1999)
4. L. Demkowicz, 2D hp-Adaptive Finite Element Package (2Dhp90) version 2.0, TICAM Report 02-06, The University of Texas at Austin (2002)
5. L. Demkowicz, D. Pardo, and W. Rachowicz, 3D hp-Adaptive Finite Element Package (3Dhp90) version 2.0: The Ultimate Data Structure for Three Dimensional, Anisotropic hp Refinitement, TICAM Report 02-24, The University of Texas at Austin (2002)
6. J.T. Oden and S. Prudhomme, Estimation of Modeling Error in Computational Mechanics. *J. Comput. Phys.*,182 (2002), 496-515
7. N. Mahmood, G. Deng, and J. C. Browne, Compositional Development of Parallel Programs, *Proceedings of the 16th Workshop on Languages and Compilers for Parallel Computing* (LCPC'03), College Station, TX, 2-4 October 2003.
8. B. Ensink, J. Stanley, and V. Adve, Program Control Language: A Programming Language for Adaptive Distributed Applications, *Journal of Parallel and Distributed Computing* , vol. 63, no. 11, pp. 1082-1104, Nov. 2003
9. P. Newton and J. C. Browne, The CODE 2.0 Graphical Parallel Programming Language, *Proceedings of the ACM International Conference on Supercomputing*, July 1992.
10. F. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckermeier, AspectIX an Aspect-Oriented and CORBA-Compliant ORB Architecture, Tech. Report TR-I4-98-08, IMMD IV, Univ. Erlangen-Nürnberg, Sep. 1998.