The Dissertation Committee for Nasim Mahmood
Certifies that this is the approved version of the following dissertation:

Productivity with Performance: Property/Behavior-Based Automated Composition of Parallel Programs from Self-Describing Components

Committee:

James C. Browne, Supervisor

Don S. Batory

Douglas C. Burger

Yusheng Feng

Calvin Lin

Dewayne E. Perry

Productivity with Performance: Property/Behavior-Based Automated Composition of Parallel Programs from Self-Describing Components

by

Nasim Mahmood, M.S., B.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May, 2007

Dedication

To my parents, Rafique Ahmed and Jahanara Akter

To my wife, Farhana Wasik (Joya)

To my son, Zafir Abrar Nasim

Acknowledgements

I would like to thank many people who have helped me during my time in graduate school. First and foremost I must thank my advisor, Professor James C. Browne, for taking me in as a doctoral student. He has offered me all the guidance and help that I could ever ask for and more. His broad knowledge has guided my research through many challenges and difficulties. When I was frustrated with my research, he has always encouraged me. He has not only helped me become a better researcher, but also made me a better person. He has taught me the courtesy, integrity, and responsibility in research and other aspects of life.

I am also grateful to my doctoral committee, Prof. Don Batory, Prof. Douglas Burger, Dr. Yusheng Feng, Prof. Calvin Lin, and Prof. Dewayne Perry, have made invaluable contributions to my dissertation. They offered their perspectives to my research, gave feedback to my papers.

My life as a graduate student has been enriched by the interactions with my fellow graduate students, especially, the fellow students of Prof. Browne. I would like to thank, in no particular order, the following people: Fei Xie, Huaiyu (Kitty) Liu, Guosheng (Simon) Deng, Young Yoon, and Kevin Kane. Although our research topics were quite different, they have always been willing to help whenever possible. I have benefited significantly from interactions with this outstanding group of people.

This dissertation is not possible without the love, support, and encouragement from my parents Rafique Ahmed and Jahanara Akter, my brother Ashique Mahmood (Rupam), my sister Rafiqa Sharmin (Luna), and last, certainly, not least, my lovely wife Farhana Wasik (Joya). Joya is the source of my happiness, strength, and energy. She has always been there for me with her love, care, support, and encouragement.

Productivity with Performance: Property/Behavior-Based Automated Composition of Parallel Programs from Self-Describing Components

Publication No._____________

Nasim Mahmood, Ph.D.

The University of Texas at Austin, 2007

Supervisor: James C. Browne

Development of efficient and correct parallel programs is a complex task. These parallel codes have strong requirements for performance and correctness and must operate robustly and efficiently across a wide spectrum of application parameters and on a wide spectrum of execution environments. Scientific and engineering programs increasingly use adaptive algorithms whose behavior can change dramatically at runtime. Performance properties are often not known until programs are tested and performance may degrade during execution. Many errors in parallel programs arise in incorrect programming of interactions and synchronizations. Testing has proven to be inadequate. Formal proofs of correctness are needed.

This research is based on systematic application of software engineering methods to effective development of efficiently executing families of high performance parallel programs. We have developed a framework (P-COM^2) for development of parallel program families which addresses many of the problems cited above. The conceptual innovations underlying P-COM^2 are a software architecture specification language based on self-describing components, a timing and sequencing algorithm which enables execution of programs with both concrete and abstract components and a formal semantics for the architecture specification language. The description of each component incorporates compiler-useable specifications for the properties and behaviors of the components, the functionality a component implements, pre-conditions and post-conditions on the inputs and outputs and state machine based sequencing control for invocations of the component. The P-COM^2 compiler and runtime system implement these concepts to enable: (a) evolutionary development where a program instance is evolved from a performance model to a complete application with performance known at each step of evolution, (b) automated composition of program instances targeting specific application instances and/or execution environments from self-describing components including generation of all parallel structuring, (c) runtime adaptation of programs on a component by component basis, (d) runtime validation of pre-and post-conditions and sequencing of interactions and (e) formal proofs of correctness for interactions among components based on model checking of the interaction and synchronization properties of the program. The concepts and their integration are defined, the implementation is described and the capabilities of the system are illustrated through several examples.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 PROBLEM STATEMENT

Many application packages in the high performance computing domain can be applied to a wide spectrum of problems in engineering and sciences [4], [11]. These application codes must operate robustly and efficiently across a wide spectrum of application parameters and on a wide spectrum of execution environments. The properties and behavior of the program may vary widely with change of the problem or with change of the execution environment. Thus adaptability and optimization based on problem or execution environment is a highly desirable feature of these application packages. Establishing correctness of parallel structures is a difficult task. Often the implementation has to be modeled manually in a particular formal language. It would be desirable to establish correctness of the implementation without going through a manual modeling step.

The complexity of the parallel structures of these codes combined with the complexity and diversity of parallel execution environments makes predicting the performance of these programs difficult. Conventional development methods for parallel programs where a program is fully developed before its performance properties are evaluated worsen the problem.

Modern computational algorithms utilize adaptive methods where the behavior of the program may change substantially during its execution so that the performance (and accuracy) of programs optimized for the initial conditions of execution may deteriorate during execution. Common practice in development of adaptive codes is to construct them as an integrated and comprehensive package of functional modules based on common, shared data structures. These packages are usually composed of a large number of parameterized functions. A package which is robust and offers a spectrum of implementations giving efficient execution across application parameters and execution environments may be very complex and very difficult to debug and to maintain and modify. 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. Thus one has to choose between performance and productivity. This problem is aggravated by the multiplicity of and constant change in parallel execution environments. Porting across execution environments with retention of efficiency often requires effort intensive redesign and re-implementation. Finally, conventional monolithic program structures make evolution of parallel programs particularly difficult.

1.2 Innovations and Contributions of This Dissertation

This research is based on innovative application of software engineering methods to effective development of efficiently executing families of high performance parallel programs. We have developed a framework (P-COM^2) for development of parallel program families which addresses many of the problems cited above. The conceptual innovations upon which P-COM^2 is based are: (i) a software architecture specification language (ASL) based on self-describing components, (ii) a timing and sequencing algorithm which enables execution of programs with both concrete and abstract components and (iii) a formal semantics for the architecture specification language. The description of each component in the ASL incorporates compiler-useable specifications for the properties and behaviors of the components, the functionality a component implements, pre-conditions and post-conditions on the inputs and outputs and state machine based sequencing control for invocations of the component. P-COM^2 utilizes these concepts in a compiler for the architecture specification language and a runtime system which unifies direct and simulated execution and runtime substitution of components. The unique capabilities implemented by the P-COM^2 compiler and runtime system include: (a) evolutionary development where a program instance is evolved from a performance model to a complete application with performance known at each step of evolution, (b) automated composition of program instances targeting specific application instances and/or execution environments from self-describing components including generation of all parallel structuring, (c) runtime adaptation of programs on a component by component basis, (d) runtime validation of pre-and post-conditions and sequencing of interactions and (e) formal proofs of correctness for interactions among components based on model checking of the interaction and synchronization properties of the program Each of these capabilities is summarized below and detailed in separate chapters of this dissertation.

1.2.1 An Architecture Specification Language based on Self-Describing Components

A P-COM^2 self-describing component consists of one or more sequential computations written in some conventional procedural programming language and a specification written in the P-COM^2 ASL. The P-COM^2 ASL specifications for a component may incorporate information on any or all of its functionality, its non-functional properties such as performance or robustness, preconditions and postconditions and a state machine which specifies the correct sequences of invocation for stateful components. Self-describing components and the ASL are detailed in Chapter 2.

1.2.2 Automated Composition

The P-COM^2 system automates composition of parallel programs from the self-describing components sketched in the previous subsection. The meta-information associated with the components by the ASL specifications, together with the programming model enables automated composition. Given specifications of a particular instance of a program family (See Chapter 2 for a definition of a program family and an instance of a program family.), the P-COM^2 compiler searches the library for matching components and instantiates an appropriate application instance. “Smart” matching based on containment relationships among components allows closest matching rather than exact matching and thus allows program instantiation in the absence of complete domain libraries.

1.2.3 Automated Adaptation

The P-COM^2 compiler automatically adds performance monitoring code to each component. This monitored information is available to the *adapt* component type. Users of the application can put their adaptation logic in the *adapt* component and use the information collected by the monitors to evaluate the effectiveness of system execution and to determine when a component replacement is needed. Adaptation is achieved by runtime replacement of components using dynamic linking.

1.2.4 Performance Modeling and Evolutionary Development

The P-COM^2 framework allows performance modeling of parallel programs starting from the design stage. The feature is based on a unified execution model which combines simulated execution with direct execution. Users can supply a performance model of a component instead of an actual implementation and the system will include its simulated execution time with the program execution time. The network is also modeled using a performance model. The unified execution model allows execution of both the abstract performance models and concrete implementation of components in the same program. Thus development can start with all abstract components and we can see if the program can meet the performance goal without providing actual implementations. Once the abstract program meets the performance goal, users can replace abstract components with actual concrete implementation and can periodically execute the program to incrementally verify performance properties. Thus in our framework a program can evolve from abstract performance model to complete program. Performance of the program can be estimated at any stage of realization.

1.2.5 Robustness and Formal Verification

The P-COM^2 framework facilitates development of robust components through provisions in the ASL for definition of preconditions and postconditions and specification of sequencing behavior of component operations. It allows runtime verification of the preconditions and postconditions and runtime verification of correct sequencing behavior by the use of interface state machines. The preconditions and postconditions work as a contract where the component guarantees the postconditions when users meet the obligations of the preconditions. Also through the use of interface state machines correct sequencing behavior of the component interactions can be ensured and verified at runtime. Finally we have provided formal semantics of the P-COM^2 ASL which can be used to reason about component interactions. By providing the semantics of the ASL and automatically generating the semantics in the formal language Communicating Sequential Process (CSP) [44] we can formally verify the interaction behaviors of a parallel program using the CSP model checker FDR [31].

1.3 DISSERTATION OUTLINE

The remainder of this dissertation is organized as follows. In Chapter 2, we present the P-COM^2 ASL and the programming model. Chapter 3 presents automated composition in detail together with related work and a case study. Automated adaptation and its related work and case studies are discussed in Chapter 4. Chapter 5 presents details about the performance modeling and evolutionary development capability together with related work and a case study. Chapter 6 gives details on how the P-COM^2 ASL enables writing robust components and formal verification of the interaction behaviors of parallel programs. Finally Chapter 7 concludes this dissertation and discusses future research directions.

Chapter 2: Software Architectures and Self-Describing Components

This chapter informally defines and describes the elements of the P-COM^2 programming system including its programming model.

2.1 Software Architecture and Domain Analysis

A *software architecture* [76] is a representation of the set of components from which a family of applications can be built and the relationships among them which define the structures for the instances of the application family. An *architecture description language* (ADL) is usually used to specify an architecture [63]. *Domain analysis* [81] is the basis for gathering the information by which to define a software architecture. It is also the process by which a set of attributes in which the properties and behaviors of the components can be defined. Property based schemes are very well suited [76] for describing the elements of a software architecture.

Conventional ADLs separate specification of components and the relationships among components. ADLs typically provide means for specification of functional and non-functional properties of components. The relationships among components are often defined in terms of communication protocols and/or connectors (which define interactions among components). In the P-COM^2 language, all of the information defining a software architecture is captured in a set of *self-describing components*. The P-COM^2 language is an architecture description language in which relationships and connectors[1] among components are implicitly defined. Connectors are synthesized at compile time by matching property specifications and interaction behaviors when the components are composed into a program. P-COM^2 uses a property based scheme for describing components. An architecture description in P-COM^2 provides both functional

[1] A connector is an instantiation of an interaction between components

and non-functional properties of the components and can describe a component in the context of an architecture.

2.2 SELF-DESCRIBING COMPONENTS

A component is one or more sequential computations and a specification for the properties and interactions of the components. Each component, in addition to implementing one or more functions, has an associated specification which defines its properties and its interactions as well as its functional signatures[2]. We call our components self-describing components. Self-describing components are the enabling concept for all of automated composition, adaptation/optimization, evolutionary development and the formal semantics of P-COM[2]. Interaction specifications include both the interactions the component accepts and those that it initiates in order to fulfill the interactions it accepts. The properties and interactions are specified in an *associative interface* which specifies the information used for selection and matching of components, a state machine which manages the interactions with other peers and the invocation of the sequential computations and a set of pre-conditions and post-conditions which are used to insure that the components execution behavior is robust. An interaction may be initiated by an incoming message (or set of messages) or by an invocation of an operation. An interaction triggers an action which is associated with some state of the state machine. The action may include execution of a sequential computation. A sequential computation executes in run to completion mode and refers only to its own local variables and its input variables. Figure 1 shows the conceptual view of a self-describing component.

The attributes (variable domains) in which the properties and behaviors of the components are defined are derived from the domain analysis for the family of

[2] A component may implement multiple related functions.

applications and the execution environments. The set of attributes in which the properties of the components are expressed is common global knowledge for the components.

There can be multiple implementations of a component implementing the same logical functionality but with substantially different behaviors, applicability, robustness, and performance properties. A given implementation might have been optimized for a particular execution environment. A component may be a complete implementation or an abstract timing or performance model. Execution of a program which includes abstract components reports estimated computation time of the program. The invocations of other components by a given component may depend on which of the interactions it implements it is currently executing.

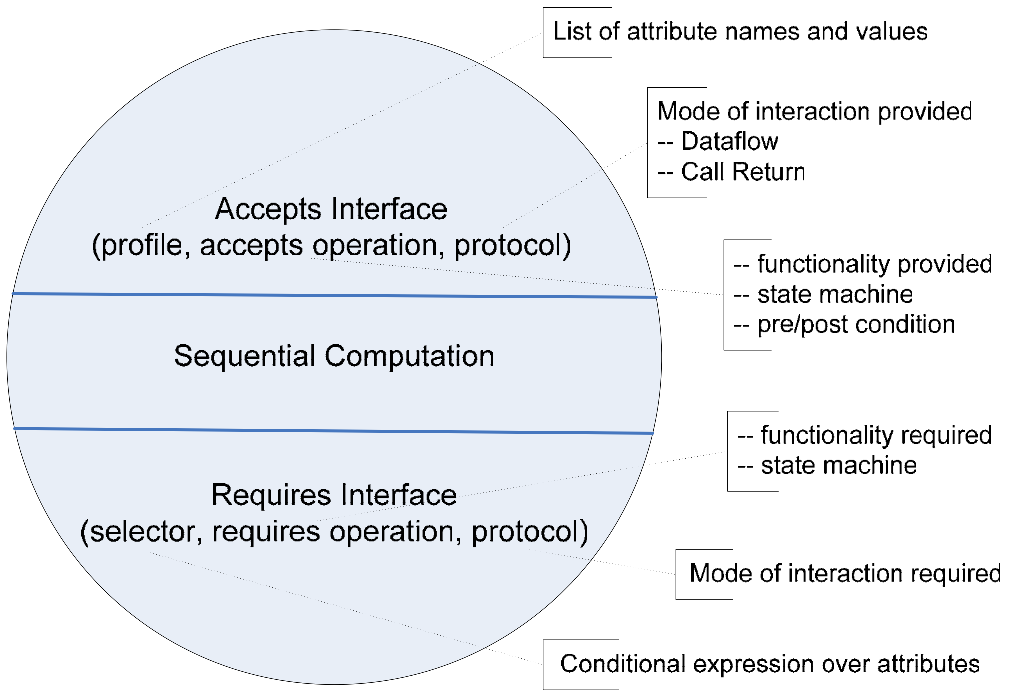
List of attribute names and values
Mode of interaction provided
-- Dataflow
-- Call Return
Accepts Interface
(profile, accepts operation, protocol)
-- functionality provided
-- state machine
-- pre/post condition
Sequential Computation
-- functionality required
-- state machine
Requires Interface
(selector, requires operation, protocol)
Mode of interaction required
Conditional expression over attributes

Figure 1: Conceptual view of a self-describing component.

The interfaces of self-describing components carry specifications for all of these properties. When a component specifies an interaction it will invoke, the invoking component will specify not only what functionality it needs, it will also specify the other non-functional properties of the required components.

Components are allowed to be stateful. The interactions of a component may depend upon its current state. Therefore invocations of the functions implemented by a component are managed by a state machine defined in the interface specifications. The state machine is defined by guards over the internal state of the component and pre-conditions and post-conditions over the inputs and outputs of the functions.

Since this information is specified in the interfaces of the components, a compiler can, given an initial condition which selects an initial component, automate the composition process by matching requirements to capabilities in libraries of components. The automated composition process is defined and described in Chapter 3.

The elements of a self-describing component together with a number of definitions that will be used in later chapters are sketched in the following.

2.3 Associative Interface

An associative interface [13] encapsulates a component. It describes the behavior and functionality of a component. One of the most important properties of associative interfaces is that they differentiate among alternative implementations of the same component. Properties of implementations such as degree of parallelism for a given component are also specified in the associative interface as runtime determined parameters. These interfaces are called "associative" because selection and matching is similar to operations on content-addressable memories. An associative interface consists of **accepts specification/interface** and **requires specification/interface**.

Accepts Interface: An accepts interface describes the set of interactions in which a component is willing to participate. The accepts interface for a component is a three-tuple (<u>**profile**</u>, <u>**state machine**</u>, <u>**protocol**</u>).

- **Profile**: A profile characterizes the properties and behaviors of a component and enables the compositional mechanism to select components meeting the requirements for efficient implementation of a given instance of an application family for a given execution environment. A profile is a set of attribute/value pairs. The attribute names and values are derived by domain analysis.
- **State Machine:** The interaction behavior of a component is managed by a state machine. Each state of the state machine is a guarded command with a condition (which is evaluated at runtime) for the execution of the function and a function signature. The state machine is defined as expressions in a linear propositional temporal logic over the attributes and state variables of the component. A function signature and its enabling condition are called an operation. An operation can be enabled or disabled based on its current state and its current state can be used in runtime binding of the components. The state machine can be used to represent complex interactions such as precedence of transactions, "and" relationships among transactions and "or" relationships among enabling states and transactions. Each operation of a component can be specified with a contractual agreement between the user of the component and the component itself. The contract is specified using pre-conditions and post-conditions. Having explicit contract of an operation helps in better understanding of the components functionally as well as automatic runtime checking of the contract.
- **Protocol:** A protocol defines a sequence of simple interactions necessary to complete the interaction specified by the profile. The most basic protocol is data-flow (continuations), which is defined as executing the functionality of a component and transmitting the output to a successor defined by the selectors at that component without returning to the invoking component.

Requires Interface: A requires interface describes 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-tuple (**selector**, **state machine**, **protocol**).

- **Selector:** A selector is a conditional expression over the attributes of the components in the domain.
- **State Machine:** State machine specifications are similar to those for accepts specifications except that the state machine is a single state.
- **Protocol:** Protocol specifications are as given for accepts specifications.

Start Component: A start component is a component that has at least one requires interface and no accepts interface. Every program requires a start component. There can be only one start component in a program which provides a starting point for the program.

Stop Component: A stop component is a component that has at least one accepts interface and no requires interface. A stop component is also a requirement for termination of a program. There can be more than one stop component of a program denoting multiple ending points for the program.

Adapt Component: An adapt component contains the logic for utilizing the behavioral information measured in the execution of the code. The fact that the measured data can be analyzed in the context of the known semantics of the components in which the measurements are taken enables straightforward analysis and decision processes.

2.4 SOFTWARE ARCHITECTURE BASED PROGRAMMING MODEL

2.4.1 Programming Model

The software architecture-based, component-oriented programming model targets development of a family of programs rather than single programs. The process defined by the programming model has two phases: development of an architecture in terms of self-describing components and specification of instances from the family of programs which can be instantiated from the set of components.

2.4.2 Domain Analysis and Component Implementation

The set of components which enables construction of a family of application programs may include components which utilize different algorithms for the same functionality for different problem instances or different implementation strategies for different execution environments. A program for a given problem instance or given execution environment is composed from appropriate components by selecting desired properties for the components and the properties of the execution environment in the Start component. The steps are:

a. Domain Analysis – Execute the necessary domain analyses to obtain the software architecture. It is commonly the case that applications require components from multiple domains.

b. Component Development – Specify and either design and implement or discover in existing libraries, the family of components identified in the domain analysis in an appropriate sequential procedural language. The specification for each component should include pre-conditions and post-conditions defining the applicability of this implementation of the functionality of the component.

c. Specify Properties and Interactions – Specify for each component (in the P-COM^2 interface definition language) its properties and the interactions in which it can engage using the attributes identified in the domain analysis to specify associative interfaces for the components. The interfaces must differentiate the components by identifying their properties in terms of the attributes defined in the domain analysis.

The resulting set of P-COM^2 self-describing components defines a software architecture for a family of application instances in which the relationships are realized at compile-time and runtime.

2.4.3 Program Instance Development

This section gives the basic process for specification of an application family instance in the case where the system configuration is known in advance and the only requirement is to compose the program from a set of components. Chapter 5 extends this process to *evolutionary development* where the system configuration is not known in advance. The steps in specifying a given instance of an application are:

a. Analyze the problem instance and the target execution environment. Identify the attributes and attribute values which characterize the components desired for this problem instance and execution environment.

b. Identify the components from which the application instance will be composed. If the needed components are not available then some additional implementations of components may be necessary together with an extension of the domain analysis.

c. Identify the dependence graph of the application instance. The dependence graph is expressed in terms of the components identified. Specify the number of replications desired for parallelism and for fault-tolerance. Incorporate these specifications into the component interfaces or as parameters in the Start component if parameterized parallelism has been incorporated into the component interfaces.

d. Define a Start component which initializes the replication parameters, sets attribute values needed to ensure that the desired components are selected and matched.

e. Define at least one Stop component.

2.5 INTERFACE DEFINITION LANGUAGE

The fundamental concepts underlying the interface definition language were given in Section 2.2 and 2.3. This section illustrates the interface of a component in the P-COM^2 syntax.

```
profile:
   string domain = "matrix";
   string function = "gather";
   string element_type = "complex";
   bool combine_by_row = true;
   bool transpose = true;
   string implementation_level = "code";
operation:
   // 1st operation
   guard { state == 0 }
   // make sure that the arguments are correct
   pre_condition { TRUE ==> (n > 0) && (m > 0) && (p >= 2);  }
   void get_p(in int n, in int m, in int p);
   post_condition { }
   action { state = 1; }

   ||

   // 2nd operation
   guard { state == 1 }
   pre_condition { TRUE ==> (inst >= 0); }
   void get_grid_n_m_inst(in mat1 grid_re,in mat1 grid_im,in int inst);
   // make sure that the values are copied into the big matrix
   post_condition { TRUE ==> forall(int i:0..(n*m-1)|
                    out_grid_re[n*m*inst + i] == grid_re[i]); }
   action { }
protocol: dataflow;
```

Figure 2: Accepts interface of gather_transpose component

Figure 2 shows the accepts interface of a component in the matrix algebra domain named gather_transpose. The function of this component is to collect the rows of a complex-valued matrix and when the collection is complete, perform a transpose of the matrix.

The accepts interface has three parts. The profile part shows the properties of this component. The semantics of the properties and their values were determined by a prior domain analysis of the program. The properties describe that this component gathers complex-valued matrices and combines them by rows and finally transposes the combined matrix. The property “implementation_level” is used to differentiate between abstract and concrete components and will be described in Chapter 5. The value of the property implementation_level describes this component as a concrete component.

The operation section shows that this component has two operations that are related by an OR (||) operation. This means that if the operations are enabled, any one of them can be invoked. The guard part of an operation decides whether the operation is enabled or not. According to the guard part, the two operations cannot both be enabled at the same time since the value of the variable “state” cannot be 0 and 1 at the same time. The initial value of the variable is 0 and thus the first operation is enabled initially. The action part of the first operation changes the value of the variable “state” to 1 after the operation is invoked. Thus the guard and action part together forms the state machine of the component. After the first operation is invoked, the second operation becomes enabled and the first operation becomes disabled.

The pre_condition and post_condition section is the implementation of the obligation and guarantee of contracts respectively which are evaluated at runtime. For example the pre_condition section of the second transaction shows the obligation of this transaction is that the value of the variable “inst” must be greater than or equal to zero. The post_condition makes sure that each individual piece of the complex matrix has been copied properly. The operations specify the parameters and their types. The protocol of the component is dataflow.

Figure 3 shows the requires interface of the gather_transpose component. The requires interface of this component has two requires clauses. Each of the requires clause shows the selector and operation part while the protocol part is omitted. The requires interface invokes a component whose desired properties are shown by the selector section of the requires interface. The first requires interface is looking for a component that can partition a complex matrix by row-wise. The second requires clause is looking for a component that can print a complex matrix. As before the guards of the operations determine which of the two operations are enabled. From the specification of the guards both operations cannot be enabled at the same time. The guard section in conjunction with the action section changes the state of the component.

```
selector:
   string domain == "matrix";
   string function == "distribute";
   string element_type == "complex";
   bool distribute_by_row == true;
   string implementation_level == "code";
operation:
   guard {  no_of_times_invoked == 1 && gathered == p }
   void get_matrix(out mat1 out_grid_re,out mat1 out_grid_im,
                   out int m, out int n*p, out int p);
   action { state = 0; initialized = 0; }

selector:
   string domain == "matrix";
   string function == "print";
   string element_type == "complex";
   string implementation_level == "code";
operation:
   guard { no_of_times_invoked == 2  && gathered == p }
   void get_grid_n_m(out mat1 out_grid_re,out mat1 out_grid_im,
                    out int m,out int n*p);
   action { no_of_times_invoked = 0; state = 0; initialized = 0; }
```

Figure 3: Requires interface of gather_transpose component

Chapter 3: Automated Composition

3.1 MOTIVATION

Component-oriented software development is one of the most active and significant threads of research in software engineering [93], [1], [21], [82]. There are many motivations for raising the level of abstraction of program composition from individual statements to components with substantial semantics. It is often the case that there is a family of applications which can be generated from a modest number of appropriately-defined components. Optimization and adaptation for different execution environments is readily accomplished by creating and maintaining multiple versions of components rather than by direct modifications of complete applications. Programs generated and maintained as compositions of components are much more understandable and thus much more readily modifiable and maintainable.

Even though there are additional benefits to component-oriented development in the distributed and parallel domain[3], there has been relatively little research on component based programming in the context of high performance parallel and distributed programming. The execution environments for parallel programs are much more diverse than those for sequential programs. It is often necessary to maintain multiple versions of parallel programs for different execution environments. Program development by composition of components enables adaptation of parallel programs to different execution environments and optimization for different application instances by replacement of components. Adaptive control of parallel and distributed programs [2] is also enabled by replacement of components. Management of adaptations such as degree

[3] CORBA, Web Services, etc. which are very much component-oriented development systems, are not commonly used for development of parallel or high performance applications.

of parallelism and load balancing are readily accomplished at the component level. Parallelism is most often determined by the number of instances of a component which are executing in parallel (single program multiple data parallelism). It has also been found that viewing programs as compositions of components tends to lead to programs with better structuring and better performance even for sequential versions.

We approach component-oriented development of parallel and distributed programs from a different perspective than most other projects. The principal concerns and goals for the P-COM^2 project have been to enable automation of composition through a compiler, to develop a mechanism enabling runtime adaptation of parallel and distributed programs at the component level [2] and to enable performance-oriented, evolutionary development of parallel and distributed programs. This chapter covers the first topic, compiler-implemented composition. Automation of composition of programs from components substantially enhances the effectiveness of component based development. In addition to the obvious benefit of programmer productivity in initial program generation, automated composition enables very rapid customization of programs to problem instances and execution environments through recompilation. Automated composition insures that interactions among components (the most commons source of error in parallel programming) are correctly generated. Perhaps surprisingly, automated composition frequently leads to programs which are more efficient that manually composed programs since compilers can generate correct code for complex behaviors such as asynchronous communication and can also recognize and generate efficient code for frequently occurring patterns of interaction behavior.

3.2 Automated Composition

The fundamental concepts underlying the interface definition language were given in Chapter 2. This section describes how the automated composition process works, shows an example of the composition process and finally shows an extension of the interface definition language that enables matching even when the program library is not complete.

3.2.1 Program Composition Process

The conditional expression of a selector is a template which has slots for attribute names and values. The names and values are specified in the profiles of other components of the domain. 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 start component which implements initialization for the program and a requires interface which specifies the components implementing the first steps of the computation and one or more libraries to search for components. The libraries should include the components needed to compose a family of applications specified by a domain analysis. The components which are composed to form a program are dependent on the requires interface of the Start component.

The compilation process first parses the associative interface of the start component. The compiler then searches a specified list of libraries for components whose accepts interface matches with the requires interface of the start component. If the matching between the selector of one component and the profile of another component is successful, the compiler tries to match the corresponding operations of the requires and accepts interface. The operations are said to match when all of the following conditions are true. 1) The name of the two operations is the same. 2) The number of arguments of each of the two operations is the same. 3) The data type of each argument in the requires operation is the same as that of the corresponding argument in the accepts operation. 4) The sequencing constraint given by the conditional expression in the accepts operation specification (the state machine) is satisfied. Finally the protocol specifications must be consistent.

The target language for the compilation process is a generalized data flow graph (GDFG) as described in CODE [69]. The GDFG has two special node types, a start node and a stop node. When compilation of the P-COM^2 Start component is completed, it is converted into a start node [69] for the GDFG and each match of a requires interface to an accepts interface results in addition of a node to the data flow graph which is being incrementally constructed by the compilation process and an arc connecting the this new node to the node which is currently being processed by the compiler. If there is a replication clause in an operation specification then at runtime the specified number of replicas of the matched component are instantiated and linked with data flow arcs. This searching and matching process for the requires interface is applied recursively to each of the components that are in the matched set. The composition process stops when no more matching of interfaces is possible which will always occur with a Stop component since a Stop component has no requires interface. Compilation of a P-COM^2 Stop component results in generation of a stop node for the data flow graph. The compiler will signal an error if a requires interface cannot be matched with an accepts interface of a desired component. The generated GDFG is then compiled to a parallel program for a specific architecture by compilation processes implemented in the CODE [69] parallel programming system.

3.2.2 Example of Composition Process

To illustrate the automated composition process, let us look at the second clause of the requires interface section of gather_transpose component shown in Figure 4. The accepts interface of the matching component, the function of which is to output the results is given as Figure 5. The P-COM^2 compiler will search its set of component libraries to find a match for the requires of the gather_transpose and generate a match with this accepts clause. This component is the Stop component and has no requires interface so the recursive matching process terminates with this component.

```
// 1st requires clause
selector:
   string domain == "matrix";
   string function == "distribute";
   string element_type == "complex";
   bool distribute_by_row == true;
   string implementation_level == "code";
operation:
   guard {  no_of_times_invoked == 1 && gathered == p }
   void get_matrix(out mat1 out_grid_re,out mat1 out_grid_im,
                   out int m, out int n*p, out int p);
   action { state = 0; initialized = 0; }

// 2nd requires clause
selector:
   string domain == "matrix";
   string function == "print";
   string element_type == "complex";
   string implementation_level == "code";
operation:
   guard { no_of_times_invoked == 2  && gathered == p }
   void get_grid_n_m(out mat1 out_grid_re,out mat1 out_grid_im,
                    out int m,out int n*p);
   action { no_of_times_invoked = 0; state = 0; initialized = 0; }
```

Figure 4: Requires interface of gather_transpose component

```
profile:
   string domain = "matrix";
   string function = "print";
   string element_type = "complex";
   string implementation_level = "code";
operation:
   void get_grid_n_m(in mat1 grid_re,in mat1 grid_im,
                     in int n,in int m);
```

Figure 5: Accepts interface of print component

To see how the automated composition process begins and continues let us examine the start component Initialize (as shown in Figure 6) for a matrix formulation of the Swarztrauber's multiprocessor FFT algorithm [92].

```
selector:
   string domain == "matrix";
   string function == "distribute";
   string element_type == "complex";
   bool distribute_by_row == true;
   string implementation_level == "code";
operation:
   void get_matrix(out mat1 grid_re,out mat1 grid_im, out int n,
                   out int m, out int p);
```

Figure 6: Requests interface of initialize component

The requires clause will be matched by a component which partitions a matrix by rows and then implements SIMD parallel computation on the partitions. Such a component is seen in Figure 7 and Figure 8. The compiler starts by matching the requires interface of the Initialize component with the accepts interface of the distribute component. The recursive process of composition is continued by the compiler seeking a matching one-D fft component to match the requires of the distribute component, and etc. This process continues until the terminating component is found as illustrated preceding.

```
profile:
   string domain = "matrix";
   string function = "distribute";
   string element_type = "complex";
   bool distribute_by_row = true;
   string implementation_level = "code";
operation:
   // make sure that the arguments are correct
   pre_condition { TRUE ==> (n > 0) && (m > 0) && (p >= 2);  }
   void get_matrix(in mat1 grid_re,in mat1 grid_im,in int n,in int m,
                  in int p);
   // make sure that the matrices ar properly copied
   post_condition{TRUE ==> forall(int i:0..(p-1),int j:0..(n_p*m-1)
                  | (grid_re[i*n_p*m+j] == out_grid_re[i][j]) &&
                    (grid_im[i*n_p*m+j] == out_grid_im[i][j]) ); }
```

Figure 7: Accepts interface of distribute component

```
// 1st requires clause
selector:
   string domain == "matrix";
   string function == "gather";
   string element_type == "complex";
   bool combine_by_row == true;
   bool transpose == true;
   string implementation_level == "code";
operation:
            int get_p(out int n/p, out int m,out int p);

// 2nd requires clause
{selector:
   string domain == "fft";
   string input == "matrix";
   string element_type == "complex";
   string fft_dimension == "1D";
   bool apply_per_row == true;
   string implementation_level == "code";
operation:
   void get_grid_n_m(out mat1 out_grid_re[],out mat1 out_grid_im[],
                     out int n/p, out int m);
} index [ p ]
```

Figure 8: Requests interface of distribute component

3.2.3 Containment Relationship and Approximate Matching

The previous sections sketched how P-COM^2 implements automated composition of programs from components by searching for a component whose accepts interface *exactly matches* the requires interface of the component whose requirements are being met. It may be the case that an exact match with the properties desired is not available in the component library. We can also specify, as a part of the architectural information, containment relationships between multiple values of a property. The component matching algorithm has been extended to implement containment relations on profile attributes. A *containment relation* can be defined for each attribute in a profile. A containment relation (A >> B) specifies that the functionality of A is a superset of the functionality of B (i.e. general purpose solver for a linear system contains triangular solver) and that A can be substituted for B if a component implementing B is not available. The requires section of a component states that it needs a component with some desired functional and nonfunctional properties. The P-COM^2 compiler searches the library of components and tries to find a component that has those properties. The search can result in an *exact match* (each desired property is found) or it can result in an *approximate match* (for some desired property, a component is found whose offered property value contains the desired property). An exact match is preferred over on approximate match. The containment relation enables composition of a program even when an exact match for a requires clause is not available.

3.3 CASE STUDY

The P-COM^2 framework has been used in the development of a number of non-trivial parallel programs. The summary results are shown below.

Linear Systems Solution by Fast Multipole Algorithm: Development of a parallel version of the matrix formulation of the fast multipole (FMM) algorithm for solution of linear systems was used to motivate and test the P-COM^2 compiler. A surprising result of this case study was the first observation that the serial version of the program composed from self-describing components was significantly faster than the serial version of the original monolithic code which was claimed to be highly optimized. The case study also showed good parallel speedup. This case study is described in Section 3.3.1.

Sweep3D: The most extensive set of experiments is based on a conversion of the DOE ASCI benchmark program, Sweep3D to self-describing components. It was found that after the rather laborious conversion to components was completed, that a pair of undergraduate students was able to generate near optimal versions of the Sweep3D code for multiple execution environments with only about two weeks of effort. It was found, as for the FMM code, that the serial version of the componentized program was significantly faster that the serial version of the original Sweep3D program. We believe that the speedup of the componentized program over the original program is due to the facts that the C compiler generates more efficient code for the relatively small code units of the components than for the complex structures in the original code. In addition good parallel speedup was observed. A full report can be found in [98].

3.3.1 Case Study – A Generalized Fast Multipole Solver

The Fast Multipole Method (FMM) [37], [38] which solves the N-body electrostatics problems in O(N) rather than $O(N^2)$ operations, is central to fast computational strategies for particle simulations. The FMM is also useful for iterative solution of linear algebraic equations associated with approximate solution of integral equations. There the FMM is used for O(N) matrix-vector multiplication. In order to adapt the FMM for applications in fluid and solid mechanics, the classical electrostatics problem must be replaced with a generalized electrostatics problem [32], [33]. Such problems involve vector and tensor valued charges, which means that one generalized electrostatics problem is equivalent to several classical electrostatics problems, which share the same geometry. In particular, FLEMS code [32] relies on the generalized electrostatics problem that is equivalent to 13 classical electrostatics problems.

We have performed a domain analysis for the FMM for generalized (multiple charge type) electrostatics. For example, the FMM tree has certain attributes, such as its depth and its number of charges per cell and the application component has an attribute with values that select between classical and generalized electrostatics. For generalized electrostatics the number of charge types is an attribute. For each attribute, the analysis defines a range of legal values. Components for a family of FMM codes for generalized electrostatics were derived from the FLEMS FMM implementation. These components were given associative interfaces that define their properties and behaviors and were annotated with domain attributes and architectural attributes. An instance of the component family can be specified by providing specific values for each attribute. An example of an attribute that would lead to different implementations is the number of charge types to be processed simultaneously.

There are space-computation tradeoffs which can be applied in the matrix-structured formulation [90] of the FMM algorithm which can be chosen to optimize the code for a given execution environment and problem specification. These include:

- Simultaneous computation of cell potentials for multiple charge types.
- Use of optimized library routines for vector-matrix multiply.
- Use of optimized library routines for matrix-matrix multiply.
- Loop interchange over the two outer loops to improve locality (within a component).
- Number of terms in the multipole expansion.

There are many variants of these structures and interactions among them. The original FMM implementation in the FLEMS code is approximately 4500 lines in length with the logic distributed throughout the code. Manual construction of optimized versions for even a modest number of execution environments would lead to rather complex code. But a small number (eight) of components characterized by the number of charges which are simultaneously computed and the number of terms in the multipole expansion suffice to realize an important subset of execution environment optimized codes.

The FMM includes five translation theorems:

- Particle charge to Multipole (P2M is applied at the finest partitioning level)
- Multipole to Multipole (M2M is applied at all partitioning levels, from the finest to the coarsest)

- Multipole to Local (M2L is applied at all partitioning levels)
- Local to Local (L2L is applied at all partitioning levels, from the coarsest to the finest)
- Local to Particle potential and forces (L2P is applied at the finest partitioning level)

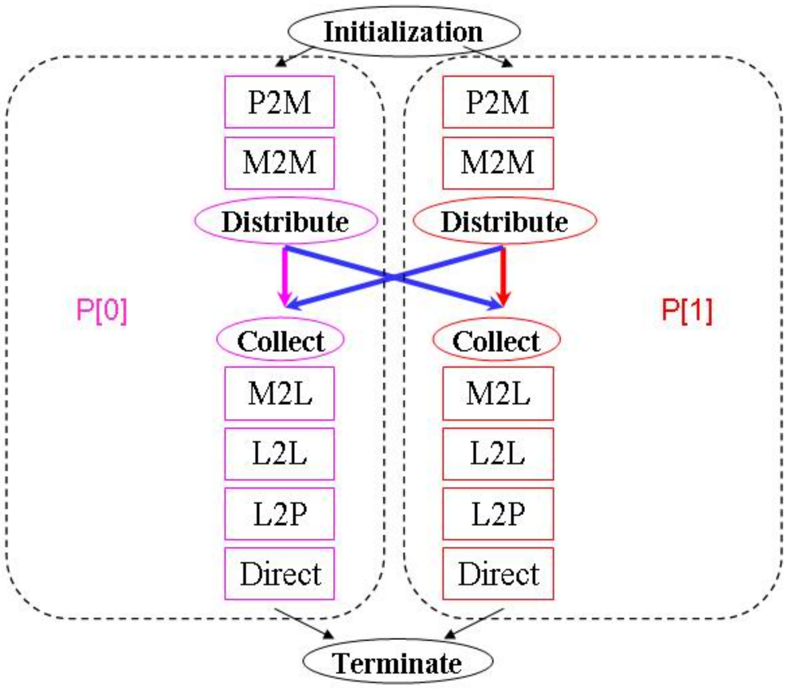

Figure 9: Data flow graph of FMM code

Two kinds of components are needed structure the FMM computation framework. The first category comes directly from the FMM algorithm. The five translation theorems, charges-to-multipole, multipole-to-multipole, multipole-to-local, local-to-local, local-to-potential and force, and direct-interaction calculation belong to this category. The second category contains the communication components, distribute and collect which actually also derive from the FMM algorithm since they implement distribution and collection according to the interaction lists for each partition of the domain. The data flow graph for the FMM code for two processors is shown in Figure 9.

Table 1. Performance data for tree depth of four.

Number of Charge Types	Run time on 2 processors (Seconds)	Run time on 4 processors (Seconds)	Run time on 8 processors (Seconds)
5	413.84	215.52	121.11
12	561.53	305.50	254.14

An extensive set of performance studies were made comparing the original and componentized sequential codes. Preliminary results were reported [27] and more detailed results were reported in [59]. The performance of the sequential componentized code, contrary to conventional wisdom, is up to 15 times faster than the original implementation which had itself been optimized by several generations of students and post-doctoral fellows. This surprising result is largely due to specialization of functionality based on selection of optimal components and replacing loop implementations of matrix-matrix multiply by BLAS implementations of matrix-matrix multiply. Table 1 shows a small sample of the performance data obtained. The data was taken on a Linux cluster of Pentium III's at 1.8 Gigahertz and a 100MB Ethernet interconnect. There are approximately half a million charges in this system. There are two factors to be noted: (i) Speedup is near-linear for the small number of processors and (ii) the time increases less than linearly with the number of charge types due to the change due to optimizations local to components.

3.4 RELATED WORK

The related work can be categorized into different categories which are described below.

3.4.1 Component-based development

COM [65], EJB [89], and CORBA [70] are the most widely used industrial component models. However they do not provide automated composition facilities and are not feasible for high performance computing.

Piccola [1] is a composition language for components. Component implementation and composition are separated in Piccola. It uses one central script which composes different components. Whereas the composition occurs during compile time in P-COM[2] using the information that is distributed among components and it is fully automated.

In the CoML [15] approach of composing components there are two parts. One is CoPL (Component Plan Language), which is basically a description of composition. The Application programmer processes these CoPL plans with a generator. The generator produces CoML (Component Markup Language) code, which can be used by different IDEs for different component technologies. The Component Markup Language is an XML application for composing software components. So this is another script based component composition where the composition is done in a central place.

H2O [91] is a component-oriented framework for composition of distributed programs based on web services. Triana [94] is a graphical development environment for composing distributed programs from components targeting peer to peer execution environments. The G2 [50] composes distributed parallel programs from web services through Microsoft .Net. Armada [73] composes distributed/parallel programs specialized to data movement and filtering.

The Common Component Architecture (CCA) project [10] is a major research and development project focused on composition of parallel programs from components. However, the goals of CCA are rather different from the goals of this project. One primary goal of CCA is to enable composition of programs from components written in multiple languages. To this end BABEL [51] has been introduced which acts as the interface specification language and uses intermediate object representation to automatically translate from one language to another. CCA has developed interface standards. The implementations of the CCA interface specifications are object-oriented. There are several frameworks including Ccaffeine [9], XCAT [35], SCIRun2 [99] and DCA [14] implementing the CCA interface specification system. The different implementations target different architectures and adopt different programming models. For example Ccaffeine targets parallel architectures and adopts a single program multiple data (SPMD) model, XCAT targets distributed architectures and adopts the grid model, SCIRun2 and DCA targets both distributed and parallel architectures and implement both SPMD and MPMD (multiple program multiple data) models. Component composition process is either graphical or through scripts and make files. CCA components interact through two types of ports. The first type of port is the provides port. The provides port is an interface that components provide to other components. The second type of port is the uses port. It is an interface through which components connects with other components which they require. These port type exhibit some similarities to the accepts and requires operation specifications. However, the details and implementations are quite different as we have focused on incorporation of the information necessary to enable composition by compilation. Users are responsible for implementing communication between replicated components which is not handled by the framework of CCA. Also diagonal communication among two different components is not defined in the CCA standard.

3.4.2 Composition Techniques

Broadway annotational compiler [40] uses annotations for retaining domain specific semantics information. Using the information the compiler can choose domain specific optimization techniques. Using dynamic feedback techniques the compiler can choose dynamically the best implementation from multiple versions of optimized code. PCOM^2 also uses semantic information in the form of attributes and their values. Using the same type of semantics information the PCOM^2 compiler can choose the best component at compiler time. The use of dynamic loading also enables our compiler to choose the best implementation at runtime which will be discussed in next chapter.

Amphion [88] is a system that uses deductive composition mechanism to automatically generate program from a subroutine library given a program specification. In order to develop a program a theory is needed for the application domain which is specified in the form of application domain axioms. The information about subroutines is also put in the forms of axioms in that domain theory. Finally there is a graphical interface that helps user to formulate the specification of the required program. The properties of this graphical construct are also put in the form of axioms. Given a formal specification of the program (using the graphical interface), the specification of the program is translated into a theorem and then a constructive theorem prover is used. The theorem prover constructs a proof showing that the goal is achievable and how to achieve it. From the given a proof a program is constructed out of the subroutines automatically.

A semi-automatic composition technique for web services is described in [87]. It has two basic parts, a composer and an inference engine. The profile of a web service has two parts – functional properties and non-functional properties. Functional properties are expressed using Web Ontology Language [22] (OWL) and have inheritance concept using OWL class. Non-functional properties describe the services. Users can add properties to the class description using DAML-S [8] which attaches semantics information to the profile of the web service. Non-functional properties are used to filter when choosing a particular web service. The idea is to start by choosing one of the web services that is registered in the composer, apply query on that service to find out what other web services it needs to implement its functionality. The composer comes back with a list of web services that can connect with the input of the selected web service. The same procedure is applied recursively to each of the selected web service. The selection from the list is manual. To make this process fully automated, AI planning techniques can be applied [96].

The ICENI [41] approach for grid services also uses OWL to annotate interfaces. This approach introduces an abstraction layer named metadata space on top of the grid services. Semantic annotation is used to describe the service as well as to describe the service methods. The meta-services use this ontological annotation to find appropriate matches between requirement publisher and implementation publisher. The semantic annotation used to describe the structure of the service method is also used to filter out incompatible matches.

ArchJava [5] annotates ports with provides and requires methods which helps the programmer to better understand the dependency relations among components by exposing it to the programmer. The accepts and requests interface of a P-COM^2 component incorporate signatures as do ArchJava provides and requires. The accepts and requests interfaces also include profiles and precedence specification carrying semantic information and enabling automatic program composition. The attribute name/value pairs in profiles are used for both selecting and matching components thereby providing a semantics-based matching in addition to type checking of the matching interfaces.

The use of associative interface has been reported earlier in the literature. Associative interface is used in one broadcast based coordination model [17]. This model uses run time composition, whereas our approach uses compile time composition.

Associative interfaces have also been reported in composition of performance modeling [16].

3.4.3 Architecture Description Languages (ADL)

Darwin [56] is a declarative binding language which can be used to define hierarchical compositions of interconnected components through programmers writing compositional scripts. It is particularly useful for describing distributed system architectures. It does not support the specification of non-functional properties. Both Darwin and P-COM^2 uses implicit connectors. In P-COM^2, the composition information encapsulates the components themselves; as a result the compiler can choose the required component automatically.

Wright [6] uses explicit connectors in describing the architecture. It uses protocol description for specifying the order of interactions between components. The composition process of specifying the attachments of a port with a role is manual. In Wright the port-role compatibility analysis is done statically. The matching of selector and profile in P-COM^2 can be seen as a kind of compatibility analysis which is done during compile time.

C2 [61], [62] is an ADL suitable for describing architectures of highly-distributed, evolvable, and dynamic systems. Component invariants and operation pre- and post-conditions are specified in 1st order logic.

Weaves [36] are networks of concurrent components that communicate by passing objects. It allows automatic composition of programs by giving the high level goals to the weaver. Component selection and interconnection is done by the weaver starting from the output goal and working backwards recursively.

UniCon [83] is an ADL with a focus on interconnecting existing components using common interface protocols. Components specify players through which they interact with outside world. Connectors (via protocols) specify roles at which the connector can mediate the interaction among components. UniCon does not support automated composition.

The SOFA environment [78],[49] describes application architecture using the SOFA component definition language (SOFA CDL). The SOFA CDL is then mapped into C++ which is used to implement the components. A component in SOFA consists of a component frame and a component architecture. A component frame lists all the interfaces that the component requires and provides and is used a black box view. A component architecture implements the operations of the provided interfaces using only internal operations and the operations of the required interfaces. A component architecture can be primitive or composed and provides a grey box view. The binding between component is explicit and manual. Whereas in P-COM^2 the compiler uses the information distributed among the components to instantiate the architecture and bindings and it is an automatic process. The connectors of SOFA are pregenerated using CORBA and dynamically linked with the components. Whereas in P-COM^2 the connectors are pregenerated using MPI.

Chapter 4: Dynamic (Runtime) Adaptation

4.1 Motivation and Overview

The need for runtime adaptation comes from two factors:

a. Adaptive computational methods may change the behavior of the program substantially during its execution. These behavioral changes may result in deterioration of performance and/or failure to meet specifications for accuracy.

b. The resource sets available to a program may change during execution leading to either deterioration of performance or opportunity for enhanced performance.

The self-describing component model in PCOM^2 enables runtime adaptation to respond to behavior changes through replacement of components and expansion or contraction of resource usage by increasing or decreasing the number of replicas of a component in the application architecture. Since components are the unit of work, composition and architecture description in our model, making components the unit of replacement and/or replication fits well within the model. In P-COM^2 a component is not loaded until it is first executed and the component interfaces have built in state machines which has the ability to enable or disable component invocation at runtime. Thus we can achieve a dynamic architecture by replacing, enabling or disabling the components which compose an application architecture at runtime.

During compile time, a search is made for a component matching the requirements specified in the self-description of each component which invokes other components. A component which meets the requirement as it is known at compile time is composed into the program. If component requirements changed during runtime, a suitable component implementation meeting the new requirements can be loaded. The behavior (performance or other property) of each component may be (selectively) monitored by the runtime system. The monitored data may be analyzed by the runtime system or sent to an *adapt* component which analyzes the data. If it is determined that some requirement is no longer being met by the currently loaded component implementation then a suitable component implementation meeting the new requirements can be loaded.

An architecture of self-describing components also simplifies load balancing and responses to changes in resource availability since increasing or decreasing the number of copies of a component running in parallel is straightforward.

The principal restriction on runtime adaptation in P-COM^2 is that the component structure of the application architecture established at compile time cannot be changed at runtime. However the degree of parallelism can vary at runtime and components can be replaced at runtime. In summary, the implementations of the components within the architecture and the number of replicas of a given component can be adapted at runtime.

4.2 IMPLEMENTATION

Most operating systems enable runtime linking of components to executable images. The requirements for intelligent use of this capability are: to identify components (through monitoring of execution behavior) which need to be replaced, to specify the properties of the component which is to be substituted for the existing component and invoke the operating system functionality to load the new component.

Composition of a program from self-describing 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 compiler automatically generates the communication path to send the monitored data to the *adapt* component. The required analysis and actions is provided in the *adapt* component or components. The analysis code in the adapt components must be provided by the programmer.

When an *adapt* component detects a need to replace a component and determines which component implementation should be used, the requires interface of the component which invokes the component for which the implementation is to be replaced is modified to reflect the current requirements. An adapt component invokes the runtime system to complete the identification of a component which meets the new requirements and then uses the operating system facilities for dynamic linking to recompose a new version of the program with the component meeting the new requirements. Thus the compile time mechanisms for program composition are extended to runtime. This unification of compile and runtime composition enables automated adaptation through a single mechanism once the programmer has provided the analysis logic to determine the adaptation to be made.

The number of replicas of a component to be executed in parallel within an application architecture is determined by parameters which can be modified at runtime thus enabling increases or decreases in parallelism at runtime.

4.3 CASE STUDY

An h-p adaptive finite element code [24] was used for the case study. The code was chosen since it is an application which may benefit from both customization at compile time and optimization at runtime. An h-p adaptive finite element code may adapt both the mesh spacing and the approximation function for the elements on a local basis in order to attain a given accuracy in the solution. (h is mesh spacing and p is the degree of the polynomial approximation to the solution on the elements of the mesh.) An h-p adaptive finite element code is therefore a good example of an application where the execution behavior may change material as it executes. The adaptive code may make many cycles through the basic loop of solution adaptation. The requirements of the solution process may change substantially as the solution mesh and approximating functions are locally or globally adapted. In a parallel implementation, the amount of work in different partitions may become unbalanced during runtime even if the initial load balance was even across processors.

The component-composition approach to application family development enabled substitution of components implementing different algorithms during execution to adapt to the changes in solution process. The case study demonstrated the effectiveness of the runtime adaptation capability. A factor of nearly three in performance was obtained through runtime replacement of the linear solver component as the solution was adapted.

The case study is based on an h-p adaptive finite element code structure developed in [24], [25], [26]. 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.

4.3.1 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 10 is a workflow diagram for a family of codes implementing *h-p* adaptive codes. Figure 10 and the components in Figure 10 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 10 because component extraction by reverse engineering of the existing code was laborious and because this componentization enables practical experiments in componentization.

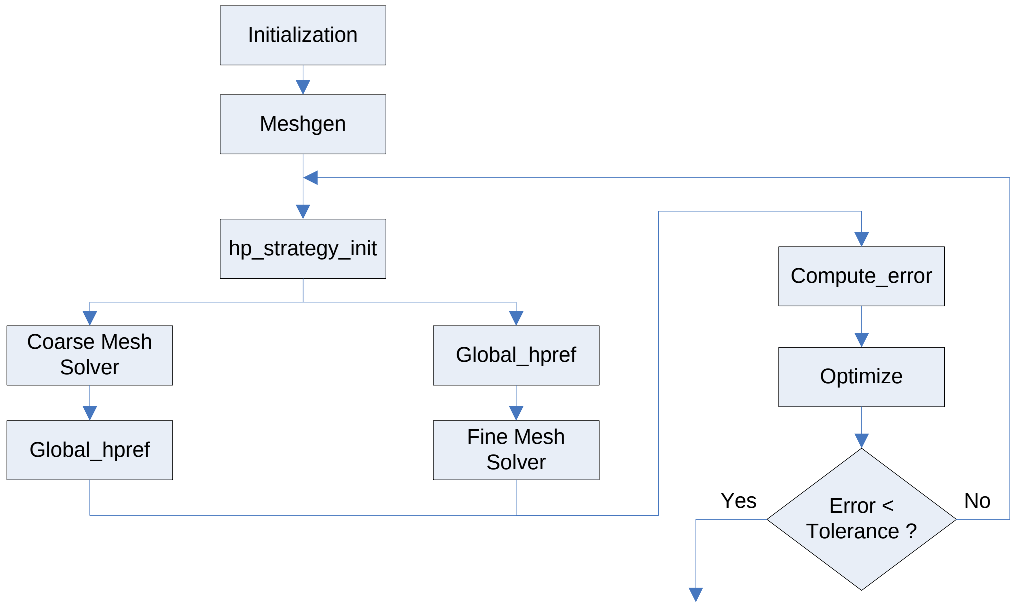

Figure 10: Workflow diagram for h-p adaptive finite element code

4.3.2 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 pre-conditioner 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 [26]).

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

4.3.3 Illustrations of Automated Composition

4.3.3.1 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 pre-conditioner. 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 P-COM^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 here).

```
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 Chapter 3. By changing the selector section of a component the appropriate implementation can be chosen at compile time.

Table 2 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 2 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 2. Properties of solutions from multiple models

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

4.3.3.2 Runtime Optimization by Component Replacement

The P-COM^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 P-COM^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.

Table 3. 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.

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 3 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.

4.4 Related Work

AspectIX [42] offers the ability to replace an implementation at runtime. The functional and configuration interface in AspectIX is similar to the operation and attributes of the profile in P-COM^2. The operation provides the syntax of a component invocation and the attributes expresses the semantics in the program domain. AspectIX uses interface information at runtime whereas P-COM^2 integrates both runtime and compile time composition.

The emerging field of autonomic computing (see [74] for a survey] is concerned with runtime adaptation of systems to evolving environments. Automate [3], [75] is an autonomic system designed to handle the complexity, heterogeneity and dynamism of grid computing environment. It features a component-based development framework to support the development of autonomic self-managed applications. Each autonomic element is controlled by an element manager/rule agent and has three kinds of ports: functional ports, control ports and operational ports. The functional ports are similar to the signature in our operation description. The control port is used to get information from sensors and to control those sensors. The operational port is used to inject interaction and behavioral rules into the component. The attributes in the profile description of our components are used in selecting the behavior of a required component and the selection mechanism is carried out by the compiler at compile time and by the runtime system at runtime. Also the interaction rules are similar to the state machine description of our operation. In case of automate a workflow is submitted to the composition manager which transforms it into a set of interaction rules and sends them to each individual element manager/rule agent. In our case the transition of workflow to state machine description is performed manually and inserted into the interface components.

COMPAS [28] is a framework for automatic performance tuning of component based systems. The monitoring and diagnosis module is responsible for acquiring runtime performance information on software components, as well as on the software application's execution environment. For that purpose it automatically instruments EJB with a proxy layer. The performance monitoring probes can use either a collaborative approach in diagnosing performance problems and in adapting the application or can use a centralized approach by sending monitored information to a central monitoring dispatcher. Adaptation functionality is based on the usage of multiple, functionally equivalent component implementations, each one optimized for a different running context. A rule based decision making process is used in selecting and activating the optimal component implementation in the current running context. P-COM^2 uses a rule based system in decision making, depends on multiple implementations, and uses a centralized approach (adapt component) in the decision making process. But it is also possible to use multiple adapt component to collaborate in the decision making process.

The ICENI [41] approach uses semantic annotation in the interface. There are two stages of semantic annotation. In the first stage the semantic annotation is used to describe the service. In the second stage the annotation is used to describe the structure of the service methods. The meta-services use this annotation to find appropriate matches. It can match semantically equivalent but syntactically different services by adapting the interface of incompatible matches based on some graph transformation rule. Thus it supports adaptive interface for composition. But it does not support adaptive components at runtime.

Adaptive MPI (AMPI) [45] is an MPI implementation and extension that supports processor virtualization. AMPI builds on top of CHARM++ [48], shares the runtime system with it, and provides the capabilities of CHARM++ in a more traditional MPI programming model. AMPI implements virtual MPI processes (VPs), several of which may be mapped to a single physical processor. It encapsulates each VP within a user-level migratable thread implemented as a Charm++ object. By embedding each thread with a chare, AMPI programs can automatically take advantage of the features of the Charm++ runtime system (such as automatic adaptive overlap of communication and computation and automatic load balancing) with little or no changes to the underlying MPI program. AMPI thus allows automatic optimization with the use of migratable threads. However it does not allow replacing components at runtime to provide better performance nor does it allow changes in the application structure at runtime. P-COM^2 supports dynamic load balancing by changing number of replicated components at runtime.

ArchJava [5] provides the ability to dynamically add components at runtime using the "new" operator, but an addition of new connection is restricted by connection patterns. These patterns define through which interfaces and to which types of components the new component can be connected. It does not provide a performance monitoring ability which can be helpful in making the decision as to when to add new components or connectors.

Darwin [57] supports constrained changes in the architecture at runtime (constrained dynamism) by replication of components via dynamic instantiation, as well as deletion and rebinding of components by interpreting Darwin scripts. Rapide [53] enables constrained dynamism by conditional connection, event patterns, and dynamic instantiation of components. C2 [61] supports unconstrained changes in the architecture at runtime by element insertion, removal and rewiring. P-COM^2 (our approach) supports constrained dynamism by replication of components by dynamic instantiation and also supports runtime reconnection using conditional operators of the state machine.

Dynamic Wright [7] is an extension of Wright [6] which allows dynamic adaptation of software architecture. The protocol description of Wright was modified to include special control events. Configurors, which are separate configuration programs use these control events to trigger reconfigurations. In case of P-COM^2 the same effect can be achieved by the use of the adapt components.

The SOFA/DCUP [78] framework enables dynamic replacement of a component at runtime. A component in DCUP is divided into a permanent part and a replaceable part. The interaction of SOFAnode and DCUP allows publisher of a component to dynamically update a component at runtime and usually it is done to reflect changes of version of a component. SOFA 2.0 [19], [43] enables modification of software architecture at runtime by introducing a set of reconfiguration patterns and permitting only those dynamic reconfigurations that are compliant with the patterns. However it does not provide any performance monitoring functionality which can be used in the decision making process.

Chapter 5: Performance Modeling and Evolutionary Development

5.1 MOTIVATION AND OVERVIEW

Designing and implementing parallel/distributed programs to meet performance requirements is still not an exact science. Attaining performance goals is rendered more difficult by the multiplicity of and constant change in parallel execution environments. Porting across execution environments with retention of efficiency often requires effort intensive redesign and re-implementation. Conventional development methods for parallel programs where a program is fully developed before its performance properties can be evaluated worsen the problem. Conventional parallel program structures based on partitioning of shared data across processes and threads make optimization for different execution environments and problem instances difficult.

We present a method (*Evolutionary Development*) for design and implementation of instances of families of parallel/distributed programs enabling evaluation of performance properties of parallel programs for arbitrary parallel/distributed execution environments at design time through performance modeling followed by evolution of the performance model to a production program. The performance model is an instance of the program where the computation of each component is a performance model for that concrete component (An evaluation of the execution time of the concrete component on some execution environment[4]) and communication times are estimated by parameterized performance models of the interconnection networks of the execution environment. When an instance of the program which meets performance specifications on a given execution environment is identified, then the abstract performance model components are

[4] Data element sizes are typically propagated through the abstract components and sometimes data element sizes must be computed or estimated in abstract components.

systematically replaced by the equivalent concrete components. This approach also enables ready customization of existing application instances to execution environments.

The research presented here extends the P-COM^2 framework which has previously been shown (Chapter 3) to compose programs from fully implemented components [59], to compose, execute and monitor the execution behavior of systems with both abstract (implemented as timing or performance models) components and concrete components. That is; a performance model of the program is constructed by an extended version of the compiler which is used to generate the concrete program. The key enabling insight is that combining a component-based program structure with a runtime system implementing an integration of direct execution and simulated execution enables execution of programs with components at multiple levels of abstraction in parallel/distributed execution environments.

The implementation is a compiler which generates code for implementation of an extended Lamport clock [52] and a runtime system which interprets associative interfaces and supports unified parallel/distributed execution/simulation of parallel programs composed from components at different levels of abstraction. The P-COM^2 compiler generates a parallel/distributed program as a precedence-constrained data dependence graph. Integration of execution behavior and parallel/distributed simulation is based on a formulation of parallel/distributed discrete event simulation as traversal of precedence constrained execution structures where the execution time is measured using the extended Lamport [52] clock defined in Section 5.2.2.

Evolutionary development begins with a program conforming to some instance of the application family architecture where some or all components are abstract (implemented as timing or performance models). Each component may have multiple representations at multiple levels of realization from analytical timing models to production code. Each component is encapsulated with an interface which specifies its properties and behaviors and distinguishes among different representations of a component. Performance evaluation begins with the P-COM^2 compiler composing the program with abstractly implemented components. This abstract program is executed in a desired execution environment. The performance of the program is evaluated to predict if the implementation will meet its performance goals. If the performance goal is not met then different compositions of the program can be evaluated for their performance until a suitable configuration is found. Then the concrete program is realized in this configuration by systematically replacing abstract components by concrete components.

A program instance need not be composed from either all abstract or all concrete components. A performance model of the program may include both concrete and abstract components. Execution of a program which includes abstract components reports estimated computation time of the program. Performance can be estimated at any stage of realization. This capability can used to evaluate the impact of different implementations of a component on performance at any stage of development. Further, as seen in Chapter 4, evolution can be continued by monitoring component behavior and replacing components during runtime.

The benefits of this approach include: (a) The abstract program has the same parallel structure as the concrete program thus eliminating a major source of uncertainty in the performance estimates. (b) Automation of model construction though compiler composition of performance models removes much of the tedious effort of model development, (c) The executions of programs realized with abstract components are very fast enabling exploration of a wide range of system configurations and (d) optimal choices for component instantiations and structures are known at design time avoiding wasted time and effort in re-implementing to correct performance problems.

There is an underlying assumption, which has been empirically verified in our experiments to date that the performance of parallel programs structured as data dependence graphs of components can be accurately modeled with simple timing models for the components and communication systems and analytic representation of contention for resources.

5.2 Integration of Direct Execution and Simulated Execution

This section describes how the integration of direct execution with simulated execution is achieved. A data flow graph model of execution is the basis of such integration. How the simulated execution is unified with this model of execution is also explained.

5.2.1 Data Flow Graph Model of Execution

The data flow model of parallel computation which underlies the unification of execution and simulation formulates a parallel execution as a dynamic generalized data flow graph (GDFG) which is an extension of the data flow graphs in [69]. The nodes of the graph contain the actions of the program which may include a local sequential discrete event simulator. The arcs specify the dependence relations between the actions of the programs. Execution of the program is traversal of the graph. The nodes of the graph are defined as six tuples ({input ports}, firing rule, an initialization, a computation, routing rule, {output ports}). Input ports are containers for a typed object or data structure. A firing rule is a conditional expression over the values in the input ports of the node. A node is enabled for execution when its firing rule evaluates to true. A computation is the action associated with the node. The routing rule of a node assigns values to the output ports of a node as soon as the computation has completed an execution. A node once enabled remains enabled until the enabled execution begins. The execution of a node is run to completion. The arcs of the graph are infinite fifo queues which bind output ports of a source node to input ports of sink nodes. Execution of a program is accomplished by generation and traversal of the directed graph. The data flow graph explicitly specifies the valid execution sequences for the components including which components can be executed in parallel.

5.2.2 Unification of Simulated Execution and Direct Execution

This section presents a data flow formulation of parallel/distributed discrete event simulation for simulation modeling of parallel/distributed systems which are formulated as precedence-constrained dynamic generalized data flow graphs and the integration of this formulation of parallel/distributed discrete event simulation with direct execution.

Sequential execution of discrete event simulation can be viewed as the generation and traversal of a dynamic, ordered list of events. Parallel/distributed execution of discrete event simulation can be viewed as generation and traversal of a directed graph of events. Parallel algorithms must partition generation and traversal of a dynamic time-ordered ordered list of events into subsets while preserving a valid order of generation and graph traversal. Valid executions of parallel/distributed discrete event simulations are constrained to traversals of the directed graph that conform to an order which would result from some sequential execution.

The parallel/distributed discrete event simulation model is formulated as a directed graph of nodes where the dependence relations among the nodes are an order-preserving subset of the nodes of the data flow graph of the actual system. In practice, the nodes with abstract models of the node computation are given the same firing rules as the nodes with the concrete code for the computation. Simulation time is generated by an extended Lamport clock [52] at each node in the graph. A Lamport clock is a mechanism for ordering the execution of events in a distributed system of concurrently and asynchronously executing processes.

a. Each process maintains a local clock and communicates by sending messages time-stamped with the value of the local clock.

b. When a process receives a message, it compares the timestamp in the message to the value of its local clock and sets its clock to the larger of these values.

This insures that any subsequent actions at the receiving process will have timestamps greater than the timestamp on the most recently received message. A Lamport clock thus maintains a logical causal order among actions in a distributed system.

The extended Lamport clock which defines causality and enables integration of actual execution and distributed simulation in the execution of the dataflow graph model of a parallel/distributed software system is defined as follows.

- An arc carrying the simulation time of a source node to each sink node of the source node is added (by the compiler) to the arc set of the data flow graph of the simulation model.

a. If the firing rule is an "and" over several ports, the start time for the execution of the node is taken to be the largest time among the current value of the local clock and the times associated with the data messages in the firing rule.

b. If the firing rule is an "or" over multiple ports then the start time for the execution of the node is a Lamport clock computation carried out for each invocation. The local clock for a node is updated to include the time (real or simulated) taken to execute the node computations and this local time is sent on the simulation time arc to nodes to which the node has a data output arc.

Causality is maintained in that the execution order will be an execution order which could have been generated by some serial execution of the actual system. No deadlock management algorithms (other than what is required for the actual system) are necessary. Parallel speed-up of execution of the simulation is bounded by the parallel speed-up of the actual system.

5.2.3 Example

The example application presented here is a parallel solution of LaPlace’s equation showcasing the accuracy to be expected when simple abstract performance models of components are used to predict performance of an application.

A parallel implementation of an iterative LaPlace equation solver partitions the matrix by rows or by columns or blocks. The partitions and overlapping elements (called shadow elements) are iteratively evaluated using the shadow elements as boundary conditions. The iterations are continued until some convergence metric becomes sufficiently small.

The algorithm for the LaPlace solver in two dimensions is as follows:

1. The NxM matrix is partitioned row wise into P sub-matrices and the sub-matrices are sent to the P processors.
2. The shadow rows are communicated. After the communication the topmost and bottom-most processor has a matrix of size N/P+1 x M and all other processors has a matrix of size N/P+2 x M.
3. Each processor performs a Jacobi iteration on its partition. A difference norm between the old values and the new values are calculated.
4. Each processor sends its value of the difference norm to a designated processor (“sum”) which collects the P difference norms.

5. The “sum” processor decides whether to stop the iteration process and sends the decision message to each of the P processor.
6. If a process receives a stop iteration message it sends its partition to the “gather” processor.
7. The designated processor collects all the submatrices and composes these into a N x M matrix.
8. The solution is printed.

Five components can be identified from this algorithm:

a. Distribute which performs step 1 and 2,
b. Jacobi: performs steps 2, 3,4 and 6,
c. Sum which performs step 5,
d. Gather which performs step 7, and
e. Print which performs step 8.

Figure 11 shows the data flow graph of the program in terms of the components identified. The data flow graph is shown for the case when the matrix is partitioned into three parts.

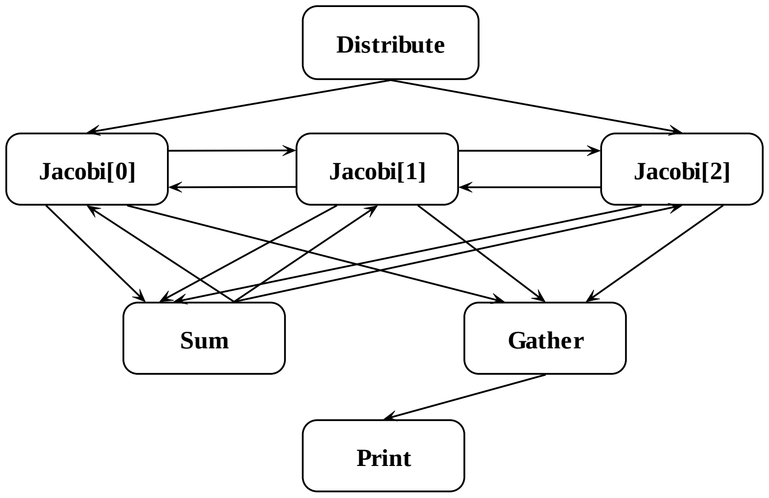

Figure 11: Data flow graph for Laplace solver

From the data flow graph, the data elements that have to be passed from one component to the other are identified. Abstract components are coded where the computation section is empty and/or is not yet implemented. The timing model for the component is added in the computation section of the abstract component to give an estimate of the runtime of the component. Communication is modeled using the size of the data elements being passed and the properties of the interconnection network. The complete program can then be run using the abstract components which gives an estimate of the runtime of the program. When the implementation of a component is complete, the concrete component can then be plugged into the program replacing the abstract component. The process of replacing an abstract component with a concrete component is continued until all the abstract components are replaced with concrete components. During the evolutionary development the estimated runtime of the program gets more and more accurate and at the end of the process we have a fully functional program.

The computational components (Jacobi and Sum) in this family of applications are floating point intensive. For these components, the computation time for each component is modeled using an estimate of the number of floating point operations needed to implement the computation. The estimated time for the computation is computed by dividing that number with the FLOPS (Floating Point Operations per Second) of the processor. Normalization of the FLOPS rate for a single component is usually sufficient to give good accuracy for computation times. The execution times for Distribute and Gather are primarily the costs for data movement and data copy which are similarly modeled with approximate instruction counts. Communication time is modeled as the expected time to send a given number of bytes. Communication time for each message is computed as a + b*x where a is a startup time for the communication to begin, b is the data transfer rate of the network and x is the given size of the data. The parameters a and b are estimated from measurements on the execution environment to be modeled. We have tried several versions of more sophisticated performance models for both computation time communication time and have not found substantial increase in accuracy. We speculate that the success of simple performance models at the component levels giving quite accurate performance estimates at the system level is due to the fact that each component implements a relatively simple and well-understood algorithm.

Table 4: Comparison of estimated & actual runtimes

Matrix Size n (nxn)	# of partitions = # of processors	Estimated runtime (sec)	Actual runtime (sec)
1024	2	27.979618	26.04458
1024	4	15.411232	14.234831
1024	8	9.275731	8.47888
1024	16	7.051624	6.31288
2048	2	107.157538	101.566281
2048	4	57.962647	54.137176
2048	8	47.306664	44.850613
2048	16	23.367203	21.459022
4096	2	432.709424	422.8589
4096	4	223.485333	218.343156
4096	8	178.698618	172.806012
4096	16	142.53143	136.246375

Table 4 shows a comparison of the estimated runtime and actual runtime for various matrix sizes and partition sizes. The measurements were taken on “lonestar” a Cray/Dell Linux cluster at the Texas Advanced Computer Center. The estimated runtime is for the program when all the components are abstract components. The estimated runtime is within 10% of the actual runtime in most of the cases.

5.3 CASE STUDY

The case study is based on hp adaptive finite element code [24]. The workflow diagram of the program and the componentization was shown in Section 4.3.1. The solution of the linear systems for the coarse and fine mesh takes about 80%-90% of the execution time of the program. Composition of a performance "optimal" instance of the h-p adaptive code is illustrated by choice of linear solver and by determination of the appropriate degree of parallelism for the coarse and fine solvers as a function on mesh properties. (“Optimal” means the lowest execution time which can be obtained using the members of the component library.) There are several choices of implementations which may have substantially different performance. The componentized structure naturally suggests executing the coarse and fine mesh solutions in parallel. The linear system for the fine mesh will have size approximately twice that for the coarse mesh. The number of diagonal bands in the matrix structure increases with the degree of the approximating polynomial. Different solution methods may be more efficient for solution of the linear systems which result from different sizes and structures for the different meshes. It may be advantageous to use a higher degree of parallelism for solution of the linear system for the fine mesh than for the coarse mesh. However, the linear system for one-dimensional finite element models is very sparse so that solution requires only modest computational work for their solution. So the overheads of communication may limit the effective degree of parallelism.

A system configuration which used concrete representations of all components except the linear solvers was executed on lonestar. For small matrices a direct solver is typically used and that was the case for the original code which we re-engineered into components. However, if the approximating polynomial is of high degree or the matrix is large, solution by an iterative method such as a conjugate gradient method can be much more efficient.

A wide range of experiments were executed ranging across mesh properties, types of linear equation solvers and degree of parallelism for the solution of the linear system from the fine mesh. Each experiment required only changing of values in requires interfaces and invocation of the compiler.

We report here the results of two experiments which lead to the important performance optimizations. The linear systems from the coarse and fine mesh were solved in parallel in both of the experiments. Each of the two experiments used an initial mesh of 500 elements with the approximating polynomial for the finite elements being chosen to be of degree 2 and degree 8. The initial linear systems for the 500x2 mesh is 1001x1001 for the coarse mesh and 4001x4001 for the fine mesh while the initial linear systems for the 500x8 mesh are 4001x4001 and 9001x9001.

Experiment 1 used an abstract performance model of the direct solver for the coarse mesh and an abstraction performance of the parallel conjugate gradient solver for the fine mesh and varied the degree of parallelism for solution of the linear system of the fine mesh. For the preconditioned conjugate gradient method it is assumed that the total number of iterations required for convergence is proportional to the square root of the spectral condition number of the input matrix. The result of experiment 1 is shown in Table 5.

Table 5: Estimated execution times for experiment 1.

Mesh (# of elements x polynomial degree)	Estimated Coarse Mesh Solution Time (sec)	Number of Processors for Fine Mesh Solution	Estimated Fine Mesh Solution Time (sec)	Estimated Total Time (sec)
500x2	0.26	1	1.65	3.08
500x2	0.26	2	8.14	9.71
500x2	.026	4	27.49	29.76
500x8	13.82	1	3.93	18.43
500x8	13.82	2	11.93	18.47
500x8	13.82	4	23.15	27.74

From this experiment we conclude that there is no performance gain from parallel execution of the conjugate gradient solver on the linear system from the fine mesh and that the direct solver is a bottleneck for larger matrices resulting from high degree approximating polynomials.

Experiment 2 replaces the direct solver for the coarse mesh with a serial implementation of the conjugate gradient solver and the parallel conjugate gradient solver for the fine mesh with this same serial conjugate gradient solver. The result of this experiment is given in Table 6.

This experiment shows that the conjugate gradient solver is only marginally faster than the direct solver for the linear systems from meshes with low degree approximating polynomials but dramatically faster for meshes with high degree approximating polynomials.

Table 6: Estimated execution times for experiment 2.

Mesh (# of elements x polynomial degree)	Estimated Coarse Mesh Solution Time (sec)	Estimated Fine Mesh Solution Time (sec)	Estimated Total Time (sec)
500x2	0.25	1.13	2.49
500x8	0.91	3.31	6.64

These (and other) experiments suggest that a concrete configuration similar to the abstract configuration of experiment 2 would be near optimal. Table 7 shows the execution times for the program with concrete components.

Table 7: Actual execution times for optimal configuration

Mesh (# of elements x polynomial degree)	Coarse Mesh Solution Time (sec)	Fine Mesh Solution Time (sec)	Total Time (sec)
500x2	0.22	1.19	2.42
500x8	0.86	3.23	6.25

The abstract performance model of the system gave quite accurate predictions of the performance of various system configurations and lead directly to a near-optimal system configuration.

In conclusion, the case study showed evolutionary development process and also showed very good prediction (within 15% of actual runtime) of parallel program performance. The combination of a component-defined program structure where the components are self-describing and the integration of execution and simulation has enabled: (a) automated support for evolutionary development of parallel/distributed programs from abstract design or performance models, (b) prediction of the performance properties of parallel/distributed programs for specific application instances and execution environments.

5.4 RELATED WORK

The most directly related research is MPI-SIM. MPI-SIM [80] predicts the performance of existing MPI programs by using direct execution to simulate sequential blocks of code and simulates a subset of MPI core functions. The simulator can run in parallel and a conservative synchronization algorithm together with a number of optimizations is used reduce the frequency and cost of synchronizations in the parallel simulator. But the simulator assumes the existence of program implementation and cannot predict the program performance at the design stage. It can, however, accurately predict the behavior of a program across multiple parallel execution environments and has been applied to several large scale parallel programs [23].

The survey paper in [12] gives a taxonomy of some existing model based performance prediction techniques. The paper classifies existing techniques in three dimensions where the dimensions are: the integration level of the software model with the performance model, the level of integration of performance analysis in the software lifecycle, and the methodology automation degree. Using the classification criterion our work falls in the category where the performance model is the same as the software model, the level of integration in the software lifecycle falls in the software design stage and the level of automation is high.

Predicting performance of computations using user input has been discussed in [95]. The user has to predict about the performance of a component and the techniques discussed in that paper can be used in asserting the prediction.

SBASCO [29] is a skeleton based system that exposes skeleton (internal structure) of components in the interface. SBASCO uses two different kind of interface. The application view interface provides the signatures of the operations provided. The configuration view interface exposes the structure. SBASCO uses a number of predefined skeletons (or patterns) that have associated cost models. Given a set of components a configuration tool uses runtime analysis to calculate the constants of the cost model. The constants together with the cost model are then used in mapping the components to the processors and also to find out the best value for the parameters such as degrees of parallelism. SBASCO thus uses a cost model based performance prediction technique in optimizing an application. However it does not have the ability to execute cost model and actual implementation in the same application resulting in evolutionary development.

COMPAS [67] is a framework for performance management in component based systems using a model driven architecture approach. It obtains real-time performance information from a running application by inserting a proxy layer in each EJB component. It then creates UML models of the target application using information from the monitoring module. The generated models of the application are simulated with different workloads to identify design problems or poor performing components. COMPAS requires a running application and uses runtime monitoring to build the application model and thus cannot be used at the design stage. The execution model of P-COM^2 together with the integration of simulated execution enables the prediction of program performance using abstract components from the design stage.

Parallel/distributed simulation research has two main branches: conservative originated by (Chandy, Misra, Bryant) [18], [20] and virtual time or optimistic originated by Jefferson [46]. In each case the execution model is the communicating sequential processes model with asynchronous execution of distributed processes communicating by messages on one way channels. In conservative simulation, causality is maintained by restricting progress at nodes which limits effective parallelism in the simulation. In optimistic simulation, causality is maintained by a clever mechanism for detecting and recovering from breaches of causality. When multiple time scales are present in the system being simulated, rollback and restart can severely restrict forward progress. There has been much research on hybrid models of distributed simulation where processes “look ahead” to both progress beyond the time allowed by pure conservative simulation and to avoid most of the breaches of causality which might occur under optimistic execution. Bagrodia and his students [64], [97] have carried several studies which use data flow graph based “look ahead” to improve the efficiency of parallel/distributed simulation. There have been many hybrid schemes many of which are described in Fujimoto’s [34] comprehensive book.

The data flow precedence-constrained execution model used herein is different from the CSP-based execution model for distributed discrete event simulations in fundamental ways.

a. The causality preserving execution sequences for nodes are derived from the data flow graph formulation of the program.

b. The simulation clock is derived from an execution order derived from the logic of the data flow model for execution of the program rather than the simulation clock determining the order of execution.

The data flow formulation of parallel/distributed simulation is not, however, a general model of parallel/distributed simulation. It applies only to systems which can be formulated in a data flow model of execution.

Chapter 6: Robustness and Formal Verification

6.1 Motivation and Overview

The increasing prevalence of parallelism in mission critical systems coupled with the increasing role of numerical computations in control systems such as medical instruments [71], [72] makes architecting parallel computation systems and establishing the correctness of parallel computation systems a task of safety critical importance. Most errors in parallel programming arise in the design and coding of interactions (synchronization and communication) among units of computation (processes, threads or components) which are executing concurrently. While there is little hope for verification of conventionally programmed parallel computation systems, definition of parallel applications in an architecture specification language with compilable/executable semantics enables all of automated composition of parallel programs, formal verification of the synchronization and communication structure and interaction properties of parallel computation systems and efficient runtime monitoring of component interactions and synchronization.

Software architecture definition languages (ADL) [76], [63] typically define software architectures as components and connectors between components. We use the phrase Architecture specification language (ASL) rather than the usual ADL since the P-COM^2 architecture specification language incorporates specification of implementation and behavioral properties of components, enables deferral of definition of connectors to compile time and has compilable semantics. Incorporation of implementation and behavior properties and deferral of definition and realization of connectors to compile time are all extensions of conventional architecture definition languages.

The P-COM^2 ASL specifies the behaviors and implementations of components and interactions among components in a manner which enables the compiler for the ASL to automatically generate parallel program structures including connectors among components and choose components appropriate for a given execution environment and problem instance. Compiler generated parallel structures should be much more likely to be correct than manually coded parallel computation structures but there is still need for verification of correctness for the communication and synchronization of the compiled parallel programs and support for programmer defined runtime checks of interactions since the specifications for the interactions may be flawed.

This chapter reports the development and application of formal verification of the interaction and synchronization properties of practical high performance parallel programs via model checking and capabilities for generating runtime monitoring of component interactions. Verification is based on development of a formal semantics for the architecture specification language (ASL) of the P-COM^2 development system for parallel programs, translation to the language of the FDR model checker [31] and application of the FDR model checker to the verification of the interactions and synchronization behavior of programs specified in the ASL.   The critical factor enabling both formal verification and generation of efficient monitoring code is that the P-COM^2 ASL rigorously separates specifications of interactions from computations enabling specification of a formal semantics for the interactions among components.

A unique specification issue is that deferral of the realization of connectors to compile time requires that the semantics of the language be defined in two phases: *for the language itself and for the execution model for the language* since the connections between the components are not explicitly defined or realized until the compiler matches the specifications among components to generate the connectors.

Model checking verification of the properties of the interactions among components requires that the component interfaces be represented in a model checkable language. This chapter defines the semantics of the P-COM^2 ASL and execution model in terms of Hoare's CSP [44]. A translator from the P-COM^2 language to FDR extension of CSP has been defined. The representations in the FDR-extended version of CSP are verified for concurrency properties using the FDR model checker.

The P-COM^2 ASL implements features targeting increased reliability and robustness including preconditions and postconditions on inputs and outputs of the component computations, fault-tolerance by replication of components, and enhanced state machine control of operation sequencing. The P-COM^2 compiler generates code for runtime verification of pre-conditions and post-conditions and state machine sequencing.

6.2 Features of P-COM2 for Improving Reliability and Robustness

This section describes the features of P-COM^2 ASL that improves robustness and reliability of an application. Compile time semantics, executable semantics, and formal verification of sequencing behavior are presented in the following sections.

6.2.1 Preconditions and Postconditions

Since a software system is built from a set of components, the correctness and robustness of the system cannot be ensured unless we can ensure the correctness and robustness of the individual software components. A component usually offers one or more service to its users. Each service of a component is a contractual agreement between the user of the component and the component itself. A contract has an obligation to fulfill and a guarantee that it provides. Given the proper set of input the component provides the correct set of output or service. The contract requires the user of the component to meet the obligations of the contract, and when the obligation is met the component guarantees to provide the correct output. The obligation of the contract is to provide the correct set of input that the component is expecting and can process. Once the user has met the obligation of the component, the component guarantees to produce correct result.

Traditionally this contract of a service has been implicit. But an implicit contract can result in software failure and in the absence of an explicit contract it becomes cumbersome to find and fix bugs. An explicit contract can result in better understanding of the behavior of the software component. Once the contract is explicitly stated in the interface of a component, it provides a precise description of the components functionality. When the service of a component is invoked, the runtime system can automatically check if the obligation has been fulfilled before the implementation of the component is invoked. If the obligation is not fulfilled the correct result cannot be generated and some appropriate action can be taken. A range of actions are possible. The action can be to print some diagnostics information and quit the program making fault diagnosis easier and giving the user direction on what went wrong. Or the user of the component can be notified to take care of the obligation. Once the obligation is fulfilled, and the implementation of the component is invoked the runtime system can automatically check if the guarantee of the component has been fulfilled by producing the correct result. If the guarantee is not fulfilled it usually means that the implementation of the component is incorrect or we have done a poor job in documenting what the component guarantees to provide. When the guarantee section of a contract fails again we can take an appropriate action. At the least we can print some diagnostic information and quit the program. Or we can invoke an alternate implementation. Invoking an alternate implementation can improve the robustness of a component.

In P-COM^2 ASL the obligation of the contract is specified as precondition of accepts operation. The guarantee of the contract is specified as postcondition of accepts operation. The precondition and postcondition together gives a precise description of the components behavior. The runtime system of the P-COM^2 compiler automatically checks the precondition before invoking the implementation and also automatically checks the postcondition after the implementation is invoked.

6.2.2 Fault-Tolerance by Component Replication

P-COM^2 ASL allows a component to be replicated. The number of replicated instantiation of a component is determined by the number of replicas specified in the requires clause of the invoking component. Replication may be done for SPMD (single program multiple datastream) parallel structuring or for fault-tolerance.

If the invocation is for SPMD parallelism then each replica will execute on different data and the component which receives the outputs of the replicated component will generally have its interface specified to receive the outputs from all of the replicas.

If the replication is for fault-tolerance, then each replica will execute on the same data and the components which receive the outputs of the replicated component will generally be programmed to receive only the output of the first successful execution of the replicated component. The receiving component will then set its state machine guard to not receive the outputs of the other replicas. Note that this replication does not require synchronization. It is also possible to collect output from all the replicated components and perform a computation such as comparison or leader election on the collected output.

It is also possible to have a requires clause which invokes MPSD (multiple program single datastream) parallelism for fault tolerance. In this case, the invoking component has separate requires clause for several different implementations of the same functionality. The receiving component will usually receive all of the components and compare the results of the several executions.

6.2.3 Runtime Verification of State Machines

The state machine specification used in the interface of the components is not only serves the purpose of specification and formal verification but also is the actual syntax of the state machine implementation. Thus it is not a model of the state machine but an actual implementation of the state machine. As a result the guards and conditions together with actions of the operations are actively monitored and verified during runtime.

6.3 COMPILE TIME SEMANTICS

The compile time semantics is presented here using tuple notation and first order logic through a number of definitions and introduction of some matching operators and component composition operator. During compile time the channels between components are established through application of component composition operators.

Component: A Component is a tuple (AI, C, RI), where AI is the accepts interface which is a set of accepts interface clause, C is the computation, and RI is the requires interface which is a set of requires interface clause. There are three types of components. A *start component* has a requests interface but do not have an accepts interface. AI is empty for a start component. A *stop component* has an accepts interface but do not have a requests interface. RI is empty for a stop component. A component is a *regular component* if it is neither a start component nor a stop component.

Accepts interface clause: An accepts interface clause AI is a tuple (P, T_A, L_A, $Indx_A$), where P is the profile which is a set of profile attributes p, T_A is a set of accepts operations, L_A is an identifier representing accepts protocol, and $Indx_A$ is an integer (greater than zero) representing optional replication parameter. In the absence of this optional parameter the value of $Indx_A$ is assumed to be one.

Intuitively the operations in T_A are related by an OR relationship so that the component can execute when any of the operations in T_A has its data ready. Whether the operation can actually execute will depend on its guard as is shown later in the execution model semantics description. In the presence of the optional parameter $Indx_A$, the input channels that are established for this accepts interface clause (as described later) will be replicated establishing *replicated input channels*.

Requires interface clause: A requires interface clause RI is a tuple (S, T_R, L_R, $Indx_R$), where S is the selector which is a set of selector attributes s, T_R is a set of requires operations, L_R is an identifier representing requires protocol, and $Indx_R$ is an integer (greater than zero) representing optional replication parameter. In the absence of this optional parameter the value of $Indx_R$ is assumed to be one.

Intuitively the operations in T_R are related by an AND relationship so that the component must try to execute all of its requires operation. Whether it can actually execute the requires operation will depend on the guard of the requires operation as will be shown later in the execution model semantics description. In the presence of the optional parameter $Indx_R$, the output channels that are established for this requires interface clause (as described later) will be replicated establishing *replicated output channels*.

Profile attribute: A profile attribute p is a tuple (t_p, n_p, a), where t_p is the type of profile attribute, n_p is the name of profile attribute, and a is the value of n_p conforming to type t_p.

Selector attribute: A selector attribute s is a tuple (t_s, n_s, Op, b), where t_s is the type of selector attribute, n_s is the name of selector attribute, Op is a comparison operator that is valid in type t_s, and b is a value that conforms to type t_s. Comparison operators = = and != are valid in every type. Comparison operator > , < , >= and <= are valid only for ordered types.

Containment relationship: A containment relationship is a tuple (t, n, a, b), where t is the type, n is the name, a is a value of type t, b is a value of type t. We say that value a contains value b. The relationship is transitive. Thus if we have a contains b, (t,n,a,b) and b contains c, (t,n,b,c) we can infer that a contains c, (t,n,a,c).

Accepts operation: An accepts operation t_A is a tuple (G_A, PreC, S_A, PostC, Act_A), where G_A is the guard which is a boolean expression, PreC is the precondition which is an expression that is checked before the execution of the component, S_A is a set of signature, PostC is the postcondition which is an expression that is checked after the execution of the component, and Act_A is the action which is a set of instructions.

Intuitively the signatures in S_A are related by an AND relationship requiring that all the signatures in S_A must be ready to execute for the component computation to execute.

Requires operation: A requires operation t_R is a tuple (C_R, s_R, Act_R), where C_R is the condition which is a boolean expression, s_R is a signature, and Act_R is the action which is a set of instructions.

Signature: A signature s is a tuple (N, n, a_0, …, a_{n-1}), where N is the name of the signature which is an identifier, n is a positive integer representing number of arguments of signature s, and a_i's (i = 0 … n-1) are the argument of signature s.

Argument: An argument a is a tuple (t, n), where t is the type of argument a, and n is the name of argument a.

Argument matching operator: The argument matching operator $\odot_{arg}$ takes as operands two arguments and produces a true/false value. Given arguments $a(t_a , n_a)$ and $b(t_b , n_b)$, $a \odot_{arg} b$ is true iff $t_a = t_b$, otherwise $a \odot_{arg} b$ is false.

Signature matching operator: The signature matching operator $\odot_{sig}$ takes as operands two signatures and produces a true/false value. Given signatures $c(N_c, n, a_0, \ldots, a_{n-1})$ and $d(N_d, m, b_0, \ldots, b_{m-1})$, $c \odot_{sig} d$ is true, iff all of the following are true

1. $N_c = N_d$

2. $n = m$

3. $a_i \odot_{arg} b_i = true$ for $i = 0, \ldots , n-1$.

$c \odot_{sig} d$ is false, otherwise.

Operation matching operator: The operation matching operator $\odot_{op}$ takes a requires operation as its left operand and an accepts operation as its right operand and produces a true/false value. Given a requires operation $t_R(C_R, s_R, Act_R)$ and an accepts operation $t_A(G_A, PreC, S_A, PostC, Act_A)$,

$t_R \odot_{op} t_A$ is true, iff $\exists s \in S_A \bullet (s_R \odot_{sig} s = true)$.

$t_R \odot_{op} t_A$ is false, otherwise.

Intuitively the matching of signature s_R and s means the possibility of the generation of a channel from the source component (the component where the requires operation resides) to the sink component (the component where the accepts operation resides). The channel can carry a structure whose fields are arguments a_0 to a_{n-1}. The name of the channel will be either the name of the signature or a compiler generated name such that the name of the channel is unique within the program's scope. The source component uses the channel as an output channel and the sink component uses the channel as an input channel. Whether the channel will be generated is decided by the successful matching of the requires interface clause and accepts interface clause as described later.

Attribute matching operator: The attribute matching operator $\odot_{attr}$ takes a selector attribute as its left operand and a profile attribute as its right operand and produces a true/false value. Given a selector attribute s(t_s, n_s, Op, b) and a profile attribute p(t_p, n_p, a), s $\odot_{attr}$ p is true, iff all of the following are true

1. $t_s = t_p$
2. $n_s = n_p$
3. The boolean expression (a Op b) evaluates to true.

Or

Value a contains (see containment relationship) value b, (t_s,n_s,a,b) and the operator Op is = =.

s $\odot_{attr}$ p is false, otherwise.

Selector and profile matching operator: The selector and profile matching operator $\odot_{SP}$ takes a selector as its left operand and a profile as its right operand and produces a true/false value. Given a selector S and a profile P,

S $\odot_{SP}$ P is true, iff $\forall s \in S, \exists p \in P$ • (s $\odot_{attr}$ p = true).

S $\odot_{SP}$ P is false, otherwise.

Interface clause matching operator: The interface clause matching operator $\odot_{IC}$ takes a requires interface clause as its left operand and an accepts interface clause as its right operand and produces a true/false value. If the application of the interface clause matching operator produces a true value then the operator also generates a channel as described below. Given a requires interface clause R(S, T_R, L_R, $Indx_R$) and an accepts interface clause A(P, T_A, L_A, $Indx_A$), R $\odot_{IC}$ A is true, and also generates a channel between t_R and t_A iff all of the following are true:

1. S $\odot_{SP}$ P = true
2. $\exists t_R \in T_R, \exists t_A \in T_A \bullet$ ($t_R \odot_{op} t_A$ = true)
3. $L_R = L_A$
4. Both $Indx_A$ and $Indx_R$ are not more than one.

R $\odot_{IC}$ A is false, otherwise.

Matching of the requires interface clause and the accepts interface clause generates a channel between the source and sink component for each matching between the requires operation and the accepts operation. If $Indx_R$ is greater than one then the *sink component is said to be replicated* and the source component gets the replicated output channel. Each of the replicated output channel i ends in the replicated component i. If $Indx_A$ is greater than one then the sink component gets the replicated input channel and the replicated input channel i starts at some replicated component i. If both $Indx_A$ and $Indx_R$ are equal to one then a simple non-replicated channel is established between the source and sink component.

Component composition operator: The component composition operator $\odot$ takes two components as operands and generates channel as described below. Given components a(AI_a, C_a, RI_a) and b(AI_b, C_b, RI_b), a $\odot$ b generates channel as described by the operator $\odot_{IC}$ iff $\exists R \in RI_a, \exists A \in AI_b \bullet$ (R $\odot_{IC}$ A = true), a $\odot$ b does not do anything otherwise.

The P-COM^2 compiler applies the component composition operator between each two components that exists in the program description and the result is the generation of channels between matching components as described by the component composition operator. In order to generate an executable program the program description must include exactly one start component, one or more stop component, and zero or more general components.

There are three scenarios that require special handling. The scenarios are the following:

Scenario 1: where t1 $\odot_{op}$ t2 returns true and t1 $\odot_{op}$ t3 also returns true (t2 and t3 are two different accepts operation) and the corresponding interface clause matches. This results in a compile time error and the user has to choose between the matching of t1 and t2 and the matching of t1 and t3.

Scenario 2: where t1 $\odot_{op}$ t2 returns true and t3 $\odot_{op}$ t2 also returns true (t1 and t3 are two different requires operation) and the corresponding interface clause matches and none of the definitions of t1, t2, and t3 uses index. The compiler in this case generates indexed channels between the two matching and generates different index for the two channels. The indices are used to describe the semantics of the execution model.

Scenario 3: where t1 $\odot_{op}$ t2 returns true and none of the definitions uses index, but t1 belongs to a replicated component and t2 belongs to a non replicated component. In this case also the compiler generates indexed channel names between t1 and t2 and uses a different index for each replica of the replicated component. The indices are used to describe the semantics of the execution model.

Scenario 2 and 3 results in indexed (or replicated) channels and requires separate treatment in describing the semantics of the sink component (described in section 6.4). The semantics of the source component indexed channels do not require special treatment other than the use of the index that will be supplied by the compiler to the component.

6.4 EXECUTION MODEL SEMANTICS

During execution, each P-COM^2 component is modeled as a process. The processes communicate through the channels that were generated by the application of the component composition operator during compile time. The semantics of the execution model is described in terms of these processes and channels. The semantics is presented using process algebra FDR CSP [31].

We use the following special processes in the translation rules.

ERROR = × -> STOP, where × denotes a special error event.

TERM = end -> STOP, where end denotes a special termination event.

Given a P-COM^2 specification for a program, P, let us use the notation TRAN(P) to denote the semantics of P in FDR CSP. Similarly TRAN(P,Q) takes two P-COM^2 definitions and produces a semantics in FDR CSP and so on.

If P is a P-COM^2 program composed of components A, B, and C where none of the components are replicated (as described in the definition of matching between requires interface clause and accepts interface clause) then

```
TRAN(P) = TRAN(A)~ [||] TRAN(B)~ [||] TRAN(C)~
```

Here for example TRAN(A) is the semantics of component A as defined later in this section and the operator ~ is the asynchrony operator as described in [47]. The asynchrony operator works by attaching buffer processes to each of the input and output channels of a process. The details of the ~ operator can be seen in [47]. The shared channels in the parallel composition operator are generated by the compiler and are omitted here for simplicity.

If a component B is replicated n times in the program then

```
TRAN(P)   =   TRAN(A)~   [||]   TRAN(C)~   [||]   ([||]   i:{0..n-
1}@TRAN(B)~)
```

CSP labels are used here to differentiate between replicas of replicated component B. Replicated output channel i ends in replicated component i. Similarly replicated input channel i starts in replicated component i. The proper connection of channels between components is done during compile time as part of the matching process.

If A is a component consisting of accepts interface AI, computation C, and requires interface RI then

```
TRAN(A) = TRAN(AI,C) ; TRAN(RI) ; TRAN(A)
```

Since accepts interface and components are closely related, the semantics of them are related and thus shown together. Thus TRAN(AI,C) denotes the semantics of AI and C in CSP.

If A is a start component then

```
TRAN(A)= TRAN(C) ; TRAN(RI)
```

If A is a stop component then

```
TRAN(A)= TRAN(AI,C) ; TERM
```

Given an accepts interface AI, and computation C, where the accepts interface AI consists of a set of accepts interface clause AIC_0 , … , AIC_{n-1} then

$\texttt{TRAN(AI,C) = [] i:\{0..n-1\} TRAN(AIC}_{\texttt{i}}\texttt{ , C)}$

Given an accepts interface clause AIC and computation C, where AIC is a tuple (P, T_A, L_A, $Indx_A$) as described in the definition of accepts interface clause then

$\texttt{TRAN(AIC,C) = TRAN(T}_{\texttt{A}}\texttt{, C, Indx}_{\texttt{A}}\texttt{)}$

Given a set of accepts operation T, computation C, and replication parameter Indx, where T consists of T_0, …, T_{n-1} then

$\texttt{TRAN(T,C,Indx) = [] i:\{0..n-1\} TRAN(T}_{\texttt{i}}\texttt{,C,Indx)}$

Given an accepts operation T, computation C, and replication parameter Indx, where T is a tuple (G, PreC, S, PostC, Act) as defined in the definition of accepts operation and S consists of signatures S_0, …, S_{n-1} then

```
TRAN(T,C,Indx) = TRAN(G) &
     TRAN(S0,Indx,r) -> … -> TRAN(Sn-1,Indx,r) ->
          (if !TRAN(PreC) ERROR
           else ( TRAN(C) ; if !TRAN(PostC) then
                                ERROR else TRAN(Act))
```

Given a signature S where is S is a tuple (N, n, a_0, …, a_{n-1}) as described in the definition of signature

`TRAN(S,Indx,r) = N?tuple_N` , if Indx = 1 but not scenario 2 or 3 as described in section 6.3.

`TRAN(S,Indx,r) = (N[0]?tuple_N [] … [] N[m-1]?tuple_N)`, if scenario 2 or 3 where the value m is supplied by the compiler as the index of the indexed channels.

`TRAN(S,Indx,r) = N[0]?tuple_N[0] -> … -> N[Indx-1]?tuple_N[Indx-1]` , If Index>1.

Here N is used as a channel name and tuple_N is used to represent a tuple variable whose fields are arguments a_0 to a_{n-1}. If the channel name N is not unique within the program then the compiler selects the channel name in such a way such that it will be unique within the program and the source and sink component uses the same unique channel name. TRAN(G), TRAN(PreC), TRAN(Act), TRAN(C), and TRAN(PostC) are similarly defined.

Given a requires interface clause RI where RI is a set of requires interface clause RIC_0, … , RIC_{n-1}

`TRAN(RI) = ; i:{0..n-1} @ TRAN(`RIC_i`)`

Given a requires interface clause RIC where RIC is the tuple (S , T_R , L_R , $Indx_R$),

`TRAN(RIC) = TRAN(`T_R`, Indx)`

Given a set of requires operation T, and replication parameter Indx, where T consists of T_0, …, T_{n-1} then

`TRAN(T,Indx) = ; i:{0..n-1} @ TRAN(`T_i`,Indx)`

Given a requires operation T and replication parameter Indx, where T is a tuple (Cond , S , Act) and signature S is a tuple (N, n, a_0, …, a_{n-1}),

```
TRAN(T,Indx) = if TRAN(Cond) then
               (TRAN(S, Indx, s); TRAN(Act)) else SKIP
```

`TRAN(S,Indx,s) = N!tuple_N` , if Indx = 1.

`TRAN(S,Indx,s) = N[0]!tuple_N[0] -> … -> N[Indx-1]?tuple_N[Indx-1]` , If Index>1.

TRAN(Cond) and TRAN(Act) are similarly defined.

6.5 An Example Showing Semantics

This section illustrates the semantics of a P-COM^2 program using a simple but practical example. This example application was introduced in [59]. The application solves the 2D FFT of a given matrix. A brief description of the application together with its workflow graph and interfaces are described in Section 6.5.1. The semantics of the example 2D FFT application is shown in Section 6.5.2.

6.5.1 2D FFT Application Example

Given an N x M matrix of complex numbers where both N and M are powers of 2, we want to compute the 2D FFT of the complex matrix. This 2D FFT can be calculated in terms of 1D FFTs using the Swarztrauber algorithm [92] which helps in parallelizing the application. The algorithm works by partitioning the matrix row wise (horizontally) and distributing the sub-matrices into available processors, applying 1D FFT on every row of the sub-matrix on each processor, collecting the sub-matrices to form a matrix and transposing the matrix and repeating the process of partitioning, distributing, applying 1D FFT on each row of sub-matrix, collecting and transposing the matrix. After the second collection and transposition operation we get the 2D FFT of the source matrix. This application can be described using five components. The components are, a start component INIT, a stop component PRINT, and three regular components DISTR, FFT_1D, and GATHER. The workflow diagram of the program is shown in Figure 12.

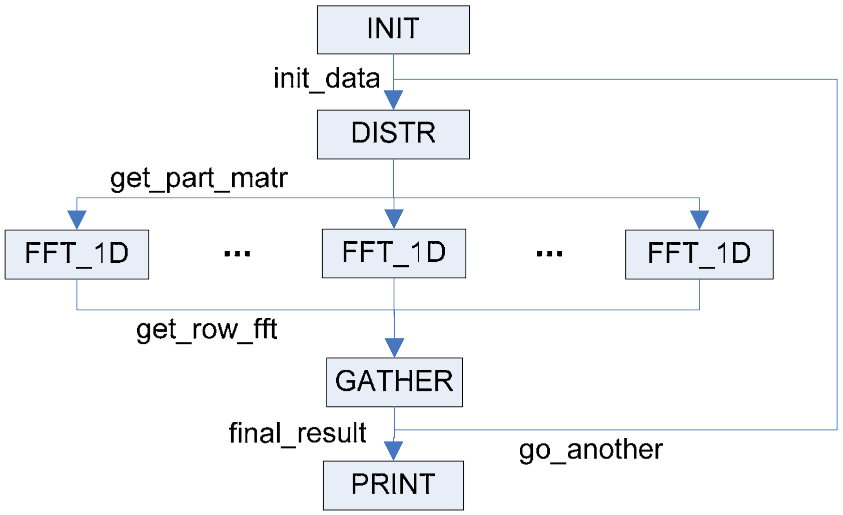

Figure 12: Workflow graph of 2D FFT application

The DISTR component partitions a matrix row-wise and sends the partition to the replicated FFT_1D components. The GATHER component collects partitioned result from the replicated FFT_1D components, transposes them and sends the result to DISTR component for the first invocation and to the stop component PRINT for the second invocation. The requires interface of INIT component is shown in Figure 13 and the accepts interface of DISTR component is shown in Figure 14. Other interfaces of the components are shown in Figures 15 through 20 (protocol is not shown, value is “dataflow” by default).

```
selector:
  string domain == "matrix";
  string function == "distribute";
  string element_type == "complex";
  bool distribute_by_row == true;
operation:
  void init_data(out mat2 grid_re,out mat2 grid_im, out int n, out int
                 m, out int p);
protocol: dataflow;
```

Figure 13: Requires interface of INIT component

```
profile:
  string domain = "matrix";
  string function = "distribute";
  string element_type = "complex";
  bool distribute_by_row = true;
operation:
  guard { got_init_data == 0  }
  void init_data(in mat2 grid_re,in mat2 grid_im, in int n, in int m,
                 in int p);
  action { got_init_data = 1; }
  ||
  guard { got_init_data == 1  }
  void go_another(in mat2 grid_re,in mat2 grid_im, in int n, in int m,
                  in int p);
  action { got_init_data = 0; }
protocol: dataflow;
```

Figure 14: Accepts interface of DISTR component

```
{selector:
  string domain == "fft";
  string input == "matrix";
  string element_type == "complex";
  string algorithm == "Cooley-Tukey";
  bool apply_per_row == true;
operation:
  void get_part_matr(out mat2 out_grid_re[], out mat2 out_grid_im[],
                     out int n, out int m, out int p);
}index [ N ]
```

Figure 15: Requires interface of DISTR component

```
profile:
  string domain = "fft";
  string input = "matrix";
  string element_type = "complex";
  string algorithm = "Cooley-Tukey";
  bool apply_per_row = true;
operation:
  void get_part_matr(in mat2 grid_re,in mat2 grid_im,in int n, in int
                     m, in int p);
```

Figure 16: Accepts interface of FFT_1D component

```
selector:
  string domain == "matrix";
  string function == "gather";
  string element_type == "complex";
  bool combine_by_row == true;
  bool transpose == true;
operation:
  void get_row_fft(out mat2 out_grid_re,out mat2 out_grid_im, out int
                   n, out int m, out int p, out int my_id);
```

Figure 17: Requires interface of FFT_1D component

```
{profile:
  string domain = "matrix";
  string function = "gather";
  string element_type = "complex";
  bool combine_by_row = true;
  bool transpose = true;
operation:
  void get_row_fft(in mat2 grid_re,in mat2 grid_im, in int n, in int m,
                   in int p, in int i);
} index [N]
```

Figure 18: Accepts interface of GATHER component

```
selector:
  string domain == "matrix";
  string function == "distribute";
  string element_type == "complex";
  bool distribute_by_row == true;
operation:
  condition { state == 0 }
  void go_another(out mat2 out_grid_re, out mat2 out_grid_im, out int
                m, out int n, out int p);
  action { state = 1; }

selector:
  string domain == "print";
  string input == "matrix";
  string element_type == "complex";
operation:
  condition { state == 1 }
  void final_result(out mat2 out_grid_re,out mat2 out_grid_im, out int
                    m,out int n);
  action { state = 0; }
```

Figure 19: Requires interface of GATHER component

```
profile:
  string domain = "print";
  string input = "matrix";
  string element_type = "complex";
operation:
  void final_result(in mat2 grid_re,in mat2 grid_im, in int n,
                    in int m);
```

Figure 20: Accepts interface of PRINT component

6.5.2 Semantics of the 2D FFT Application

This section illustrates the semantics of the P-COM^2 compiler and the execution model of the resulting program using the example that was presented in Section 6.5.1.

The component composition operator ⨀ is applied between each possible pair of components in the program description. A channel named init_data is generated from the application of INIT ⨀ DISTR. Similarly other channels are generated and are shown as annotation on the arcs of Figure 12. Let’s explain how the init_data channel is generated.

The application of interface clause matching operator $\odot_{IC}$ between the requires interface clause of component INIT (Figure 13) and the accepts interface clause of component DISTR (Figure 14) returns true because the selector and profile matching operator $\odot_{SP}$ returns true, the operation matching operator $\odot_{op}$ returns true and also the protocol matches (index's default value is one and thus do not violate the matching condition of $\odot_{IC}$). Thus application of component composition operator $\odot$ generates the channel named "init_data" which is used as an output channel by component INIT and as an input channel by component DISTR. After the compilation stage we get a number of processes and channels connecting them as in Figure 12.

```
V = { 1,2 } {- values transferred thru channels,not important since
we are modeling state machine only -}
replica_number = {0..1} {- we are modeling 2 replicas of the FFT_1D
component -}
channel from_user,init_data,init_data', go_another,go_another',
final_result,final_result': V
channel get_part_matr,get_part_matr', get_row_fft,get_row_fft' :
replica_number.V
channel end
{- the channel names and processes that end with ' are for buffering
purpose -}
BUFF(in,out,n) = {- buffer process for implementing asynchronous
operation -}
      let
         B(s) =   not null(s) & out!head(s) -> B(tail(s))
                  []
                  #s < n & in?x -> B(s^<x>)
      within B(<>)

{- the from_user channel is not in the program but introduced for
simplified property checking -}
INIT = from_user?x -> init_data!x -> SKIP
INIT' = INIT [init_data <-> init_data'] BUFF(init_data',init_data,5)
{- we are using a buffer size of 5 throughout the program for quick
checking of properties -}
```

Figure 21: Semantics of FFT program using FDR CSP syntax

The FDR/CSP program resulting from the translation is given in Figure 21 and Figure 22. The program has been manually edited to make it more readable. The reader may wish to refer to the workflow graph (Figure 12) and the ASL interface definitions when reading the FDR/CSP program. Figure 21 and 22 is literally the data flow graph resulting from unrolling the workflow graph. Note the simplicity of the state machines and small ranges for the integer variables in the state machines.

```
DISTR(got_init_data) =
   (got_init_data == 0 & init_data?x -> get_part_matr.0!x ->
                         get_part_matr.1!x -> DISTR(1))
[] (got_init_data == 1 & go_another?x -> get_part_matr.0!x ->
                         get_part_matr.1!x -> DISTR(0))
DISTR'(got_init_data) = ((BUFF(go_another,go_another',5)
      [go_another' <-> go_another] (BUFF(init_data,init_data',5)
      [init_data' <-> init_data] DISTR(got_init_data)))
      [get_part_matr <-> get_part_matr']
      BUFF(get_part_matr',get_part_matr,5))
      [get_row_fft <-> get_row_fft'] BUFF(get_row_fft',get_row_fft,5)

FFT_1D(i) = get_part_matr.i?x -> get_row_fft.i!x -> FFT_1D(i)
FFT_1D'(i) = (BUFF(get_part_matr.i,get_part_matr'.i,5)
            [get_part_matr'.i <-> get_part_matr.i]FFT_1D(i))
            [get_row_fft.i <-> get_row_fft'.i]
      BUFF(get_row_fft'.i,get_row_fft.i,5)
FFT_1D_REPLICAS' = [|{}|] i:{0..1} @ FFT_1D'(i)

GATHER(state) = get_row_fft.0?x -> get_row_fft.1?x  ->
         ((state == 0 & go_another!x -> GATHER(1))
       [] (state == 1 & final_result!x -> GATHER(0)) )
GATHER'(state) = (GATHER(state) [go_another <-> go_another']
      BUFF(go_another',go_another,5))[final_result <-> final_result']
      BUFF(final_result',final_result,5)

TERM = end -> STOP
PRINT = (final_result?x -> SKIP) ; TERM
PRINT' = BUFF(final_result,final_result',5)
         [final_result' <-> final_result] PRINT

FFT_PROGRAM = (( (  ((INIT' [|{|init_data|}|] DISTR'(0))
                             [| {|get_part_matr|}  |]
                             FFT_1D_REPLICAS' )
                      [| {|get_row_fft, go_another|}  |]
                      GATHER'(0) )
                 [| {|final_result|} |]
                 PRINT' ))
```

Figure 22: Semantics of FFT program using FDR CSP syntax (continued)

The translated program was model-checked using FDR for the following properties: 1) for every input, the program should give us an output (SPEC_1), 2) complete sequencing behavior of the operations (SPEC_2), and 3) deadlock checking (SPEC_3). The properties are shown in Figure 23. Our implementation passed all the properties. The program specification was reduced to 1365 states and FDR used 128k memory. The refinement check used 113 state with 165 transitions and took less than a second for each refinement on a 2.4GHz Pentium 4 with 1GB of memory under Debian Linux.

```
{- SPEC_1 says that for an input thru from_user channel we will get
output thru final_result channel -}
SPEC_1 = (from_user?x -> final_result.x -> STOP)
{- check that our implementation satisfies the property SPEC_1 -}
assert SPEC_1 [FD= ( FFT_PROGRAM \
{|init_data,get_part_matr,get_row_fft,go_another,end|} )

{- full specification showing the sequencing relationship of each
event -}
SPEC_2_helper(x) = ((get_part_matr.0.x -> (get_row_fft.0.x -> SKIP
            ||| (get_part_matr.1.x -> SKIP)); get_row_fft.1.x ->
SKIP))
SPEC_2 = (from_user?x -> init_data.x -> (SPEC_2_helper(x);
      go_another.x -> (SPEC_2_helper(x) ; final_result!x -> end ->
      STOP)))
{- check that our program follows the sequencing relationship -}
assert SPEC_2 [FD= FFT_PROGRAM

-- deadlock checking or check that shows that our program terminates
SPEC_3 = end -> STOP
assert SPEC_3 [FD= ( FFT_PROGRAM \
      {|from_user,init_data,get_part_matr,get_row_fft,go_another,
                    final_result|} )
```

Figure 23: Properties checked on FFT_PROGRAM

```
profile:
  string domain = "matrix";
  string function = "distribute";
  string element_type = "complex";
  bool distribute_by_row = true;
operation:
  guard { got_init_data == 0  }
  void init_data(in mat2 grid_re,in mat2 grid_im, in int n, in int m,
                 in int p);
  action { }
  ||
  guard { got_init_data == 1  }
  void go_another(in mat2 grid_re,in mat2 grid_im, in int n, in int m,
                  in int p);
  action { got_init_data = 0; }
protocol: dataflow;
```

Figure 24: Accepts interface of DISTR component with erroneous state machine

```
DISTR(got_init_data) =
   (got_init_data == 0 & init_data?x -> get_part_matr.0!x ->
                  get_part_matr.1!x -> SKIP)
 [](got_init_data == 1 & go_another?x -> get_part_matr.0!x ->
                  get_part_matr.1!x -> DISTR(0))
```

Figure 25: FDR translation of erroneous DISTR component

We artificially introduced an error in the DISTR component so that it did not change the state of the component correctly in the first operation as shown in Figure 24. The FDR translation of the erroneous state machine is shown in Figure 25. The resulted program failed to pass any of the properties and provided a trace as counter example showing why the property failed. The trace was useful in finding the bug since it showed why DISTR component was not ready to take input even though the GATHER component was ready to output. While it is easier to find errors for the simple example of this paper, for more complex systems the errors may be quite difficult to detect using informal means.

6.6 RELATED WORK

There has been research on model checking parallel numerical programs using symbolic execution [86]. The model checking approach requires that a sequential version of the parallel program be provided which serves as a specification for the parallel one and uses equivalence to establish the correctness of the parallel program in terms of the sequential one. There has been research on direct model checking of mpi programs [85], [84], [77]. MPI communication calls are represented as finite-state models abstracted from the program. As in our approach, this research verifies only the communication and synchronization properties. In P-COM^2 we represent communication and synchronization as finite state models but generate the mpi library calls during composition. Automated composition avoids the errors which can occur in manual transcription between the mpi state machines and the calls to the mpi library.

There is a substantial literature on ADLs. For a comparative study, the ADL survey paper by Medvidovic and Taylor [63] is an excellent source. We restrict our related work discussion to those ADLs for which a complete or partial formal semantics has been formulated. We categorize the related work into two categories. The related work in the first category (Darwin [55], [57], Wright [6], SOFA [79], and Rapide [36]) have complete semantics whereas (C2 [61], [62], Weaves [36], UniCon [83]) have defined only a partial formal semantics. We provide only a brief description of the related work in the second category.

Darwin [55], [57] is a declarative binding language which can be used to define hierarchical compositions of interconnected components through programmers writing compositional scripts. It is particularly useful for describing distributed system architectures. It does not support the specification of non-functional properties. It supports constrained dynamism by replication of components via dynamic instantiation, as well as deletion and rebinding of components by interpreting Darwin scripts. P-COM^2 also supports constrained dynamism by replication of components by dynamic instantiation and also supports runtime reconnection using conditional operators. Both Darwin and P-COM^2 uses implicit connectors. The semantics of Darwin is described in π-calculus [66] which allows sending of a connection name to a different component as part of a message. Darwin can [58] either use a graphical notation named labeled transition system (LTS) or a process algebra textual notation named finite state processes (FSP) to describe the behavior of individual components. The semantics of the architecture is automatically generated from the user supplied component behaviors. A tool named labeled transition system analyzer (LTSA) can be used for deadlock checking, and safety and liveness property checking. In P-COM^2 we can also check these types of properties using FDR. However in our case the FDR program can be generated directly from the program whereas in Darwin the user has to supply the component behavior. The component behavior specified in Darwin is only a model and may not be followed at runtime. In P-COM^2 the model can be generated directly from the implementation. Also the composition process in Darwin is manual whereas it is automatic in our approach.

Wright [6] uses explicit connectors in describing the architecture. It uses protocol description for specifying the order of interactions between components. CSP [44] is used for specifying the protocol descriptions in ports, roles, and glues as well as describing the semantics. FDR is also used in Wright for checking port-role compatibility as well as deadlock checking of connectors. But the composition process of specifying the attachments of a port with a role is manual. Dynamic Wright [7] is an extension of Wright to include dynamism of software architecture. The protocol description was modified to include special control events. Configurors, which are separate configuration programs use these control events to trigger reconfigurations. In case of P-COM^2 the same effect can be achieved by the use of the adapt components [60].

The behavior protocol used in SOFA [79] uses regular expressions as syntax for generating a set of traces that are permitted by a protocol. Classical regular expressions operators were enhanced by introducing operators necessary for modeling interaction of concurrent processes/agents. Interface protocols model the interaction behavior on a particular interface. Frame protocols model the interaction behavior of a component's provides and requires interface. Architecture protocols model the interaction behavior of all the components of an architecture. The interface and frame protocols are provided by the user whereas the architecture protocol is automatically generated by SOFA CDL (component definition language) compiler. The semantics of protocol conformance is explained in terms of the language described by the protocol.. Interface protocol conformance can be used to check if one interface is compatible with another interface. Frame protocol conformance with the interface protocol can be used to check if an interface is being correctly used in a component. Finally the architecture protocol conformance with the frame protocol can be used to check if the architecture will behave correctly given the behaviors of the components. The CDL compiler automatically generates architecture protocols and tests the interface, frame and architecture protocol conformance. The protocols are written separately from the SOFA executable code. SOFA thus uses protocol guard and runtime system to check if the implementation is within the constraints of the protocol guard. P-COM^2 generates the model of the sequencing behavior of the components from the actual specification of the state machine. The implementation is constrained by the state machine at runtime and thus there is no need of constructs like protocol guard for checking the sequencing behavior at runtime. P-COM^2 statically check the state machine for correctness and dynamically check the implementation for correctness by looking at the actual data values being transmitted by the use of pre and post conditions. SOFA approach uses a scripting language for program composition whereas P-COM^2 automatically composes programs from components encapsulated in its ASL.

Rapide [53] is an ADL that can be used for modeling and simulation of the dynamic behavior described by an architecture. It uses events (partially ordered event set, poset) to characterize component interaction and provides a fixed set of connector types to characterize how events flow between components. Connectors in Rapide can be modeled by defining new kinds of components and thus the connectors in Rapide are also implicit. It supports constrained dynamism by conditional connection, event patterns, and dynamic instantiation of components. The timed poset model allows modeling of non-functional property like modeling of timing. However it does not allow non-functional properties of components or connectors. The semantics of Rapide is described in terms of poset and event processing [54]. Constraints in Rapide can be used to restrict the behavior of components and can be checked at runtime for violation detection. The guards, preconditions and postconditions of P-COM^2 operations can be used in achieving the same goal.

C2 [61], [62] is an ADL suitable for describing architectures of highly-distributed, evolvable, and dynamic systems. Component invariants and operation pre- and post-conditions are specified in 1st order logic. For connectors partial semantics is specified by message filters. C2 supports unconstrained dynamism by element insertion, removal and rewiring.

Weaves [36] are networks of concurrent components that communicate by passing objects. The semantics of the components are given using partial ordering of input and output objects while the semantics of the connectors are given by the naming conventions of the queue. It allows automatic composition of programs by giving the high level goals to the weaver. Component selection and interconnection is done by the weaver starting from the output goal and working backwards recursively.

UniCon [83] is an ADL with a focus on interconnecting existing components using common interface protocols. Components specify players through which they interact with outside world. Connectors (via protocols) specify roles at which the connector can mediate the interaction among components. The semantics of the components and the connectors are implicit in their types and additionally the property list can be used to provide further semantics. UniCon does not support automated composition.

Chapter 7: Conclusions and Future Research

Parallel programming has always been a complex task. Parallel programming techniques have been typically employed in scientific computing where performance gets more priority than productivity. Although performance is very important, we cannot overlook the impact of software productivity. It has been well known that maintenance of software is the most costly part of software life cycle. The critical issue for parallel programming is to increase productivity while improving performance over the life of a family of programs. With the rise of multicore chips, parallel programming will be more pervasive so that combining productivity, parallelism and performance becomes even more important. With the increasing prevalence of parallelism and parallel computation in mission critical systems it is important that the correctness of parallel programs be established at design time and also be validated at runtime.

We presented the conceptual foundations for the P-COM^2 development environment which are a software architecture specification language based on self-describing components, a timing and sequencing algorithm which enables execution of programs with both concrete and abstract components and a formal semantics for the architecture specification language. These concepts are a synthesis from multiple disciplines of computer science including, artificial intelligence, compilers, software architecture, component-oriented development, distributed and parallel computing, and model checking.

We defined and described the compiler and runtime system which implements these concepts. The compiler composes parallel programs from independently written components; the runtime system enables monitoring and runtime adaptation at the component level. The compiler and runtime system together were shown to enable evolutionary development of programs to meet performance goals and runtime adaptation of programs by component substitution. A formal semantics for the ASL was developed. Formal verification of component interactions and state machines by translation of ASL instances to model checkable languages was formulated. Each capability of the P-COM^2 development environment was illustrated and evaluated by one or more examples

The programming methodology and tools developed in this dissertation enhance productivity by:

a. Automated composition of program instances from families of components.
b. Enabling design of instances of an application family to meet performance goals.
c. Raising the level of abstraction of program composition to the component level.
d. Enabling reuse of components across instances of an application family
e. Enabling runtime adaptation of a program at the component level.
f. Enhancing program understanding through yielding simple and clean program structures.
g. Providing a basis for better understanding of component and program behavior through precise description of the properties and behaviors of components and thus programs composed from components.
h. Runtime validation of program behaviors through preconditions and postconditions.
i. Verification of correctness of state machines and component interactions during design time.

Performance is enhanced by:

a. Design time evaluation of performance.
b. Customization of program instances to problem cases and execution environments

c. Runtime adaptation to maintain performance when execution environments or problem behavior changes.

7.1 Future Research Directions

While the P-COM^2 approach to development of parallel programs has great potential, its application is impeded by the requirement that there exists a family of components from which application instances can be composed. The parallelism which can be implemented in P-COM^2 is limited by the capabilities of the MPI and threads packages to which we compile. Additionally, we have applied P-COM^2 only at the level of functionally defined components. It could potentially also be used to compose larger systems from existing applications.

The Weaves [68] framework enables separation of global variables while composing applications from existing applications. It uses light-weight threads for connecting the applications. Much of the re-engineering effort done during componentization of legacy systems in P-COM^2 comes from removal of global variables. Integration of Weaves with P-COM^2 can substantially reduce the re-engineering cost. Also the light-weight threads of Weaves can be used to take advantage of multi-core machines. The speedup of parallel programs will be much better when we can take advantage of both clusters and multiple processors. A important practical means of enhancing both Weaves and P-COM^2 is to integrate the them.

A unification of the ASL of P-COM^2 with other modeling and software architecture tools is an important direction of research.

More case studies need to be done to see the effectiveness and scalability of the model checking technique in proving correctness of parallel programs. Translations to other model checking languages and tools to extend the applicability of model checking would be desirable. Use of P-COM^2 ASL as an annotation language in existing programs to enable automatic compilation of parallel structures and model checking of non-component based programs is in consideration.

Bibliography

[1] Achermann F., Lumpe M., et al., Piccola - a Small Composition Language, in *Formal Methods for Distributed Processing - A Survey of Object-Oriented Approaches*, pp. 403-426, Cambridge University Press, 2001.

[2] Adve V., Akinsanmi A., et al., Model-Based Control of Adaptive Applications: an Overview, in *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, April 2002.

[3] Agarwal M., Bhat V., et al., AutoMate: Enabling Autonomic Applications on the Grid, Proceedings of the Autonomic Computing Workshop, *5th Annual International Active Middleware Services Workshop* (AMS2003), Seattle, WA, USA, IEEE Computer Society Press, pp 48-57, June 2003.

[4] Ainsworth M., and Oden J.T., A Posteriori Error Estimation in Finite Element Analysis. John Wiley & Sons, New York, (2000).

[5] Aldrich J., Chambers C., et al., ArchJava: connecting software architecture to implementation, in *Proceedings of the 22nd International Conference on Software Engineering,* pp. 187-197, May 2002.

[6] Allen R., and Garlan D., A formal basis for architectural connection, *ACM Trans. Softw. Eng. Methodol*. 6, 3 (Jul. 1997), 213-249.

[7] Allen R., Douence D., and Garlan D., Specifying and Analyzing Dynamic Software Architectures, Lecture Notes in Computer Science, Volume 1382, Jan 1998, pp. 21-37.

[8] Ankolekar A., Burstein M., et al., DAML-S: Web service description for the semantic web. In *Proceedings of the First International Semantic Web Conference*, 2002.

[9] Allan B. A., Armstrong, R. C., et al., The CCA core specification in a distributed memory SPMD framework, *Concurrency Computation,* 14:1–23, 2002.

[10] Armstrong R., Gannon D., et al., Toward a Common Component Architecture for High-performance Scientific Computing, in *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation,* pp. 115-124, August 1999.

[11] Babuska I., and Strouboulis T., Finite Element Method and its Reliability. Oxford Univ. Press (2001).

[12] Balsamo S., Marco A. Di, et al., Model-based Performance Prediction in Software Development: A Survey, *IEEE Transactions on Software Engineering,* Vol 30, N. 5, pp. 295-310, May 2004.

[13] Bayerdorffer B., Associative Broadcast and the Communication Semantics of Naming in Concurrent Systems, Ph.D. Dissertation, Dept. of Computer Sciences, University of Texas at Austin, December 1993.

[14] Bertrand F., and Bramley R., DCA: A Distributed CCA Framework Based on MPI, *Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments* (HIPS'04), vol. 00, no. , pp. 80-89, 2004.

[15] Birngruber D., Coml: Yet another, but simple component composition language, in *Workshop on Composition Languages,* WCL'01, pp. 1-13, September 2001.

[16] Browne J.C., and Dube A., Compositional Development of Performance Models in POEMS, in *International Journal of High-Performance Computing Applications*, vol. 14(4), Winter 2000.

[17] Browne J.C., Kane K., et al., An Associative Broadcast Based Coordination Model for Distributed Processes, in *Proceedings of COORDINATION 2002,* LNCS 2315, pp. 96-110, 2002.

[18] Bryant R.E., Simulation of packet communication architecture computer systems. MIT:TR-188, Massachusetts Institute of Technology, 1977.

[19] Bures T., Hnetynka P., and Plasil F., SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model, *Fourth International Conference on Software Engineering Research, Management and Applications* (SERA'06), pp. 40-48, Aug 2006.

[20] Chandy K.M., and Misra J., Distributed Simulation: A Case Study in Design and Verification of Distributed Programs, *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, Sept. 1979, pp. 440-452.

[21] Czarnecki K., and Eisenecker U.W., Components and Generative Programming, in *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Springer-Verlag LNCS 1687, pp. 2-19, 1999.

[22] Dean M., Connolly D., et al., Web ontology language (OWL) reference version 1.0. W3C Working Draft 12 November 2002, http://www.w3.org/TR/2002/WD-owlref-20021112/.

[23] Deelman E., Bagrodia R., et al., Improving Lookahead in Parallel Discrete Event Simulations of Large-Scale Applications using Compiler Analysis, *Proceedings of the 15th Workshop on Parallel and Distributed Simulation* (PADS 2001), May 2001. p. 5-13.

[24] Demkowicz L., and Kim C.W., 1D hp-Adaptive Finite Element Package. Fortran 90 Implementation (1Dhp90), TICAM Report 99-38, The University of Texas at Austin (1999).

[25] Demkowicz L., 2D hp-Adaptive Finite Element Package (2Dhp90) version 2.0, TICAM Report 02-06, The University of Texas at Austin (2002)

[26] Demkowicz L., Pardo D., and Rachowicz W., 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)

[27] Deng G., New approaches for FMM implementation, Masters Thesis, Dept. of Manufacturing Systems Engineering, University of Texas at Austin, 2002.

[28] Diaconescu A., Mos A., and Murphy J., Automatic Performance Management in Component Based Software Systems, *Proc. of IEEE International Conference on Autonomic Computing* (ICAC-04), pp. 214-221, May 2004.

[29] Diaz M., Rubio B., et al., SBASCO: Skeleton-Based Scientific Components, *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing* (PDP'04), p. 318, 2004.

[30] Ensink B., Stanley J., and Adve V., 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.

[31] Formal Systems (Europe) Ltd. Failures Divergence Refinement: FDR2 User Manual, 1997.

[32] Fu Y., Klimkowski K.J., et al., A fast solution method for three-dimensional many-particle problems of linear elasticity, in *International Journal for Numerical Methods in Engineering*, vol. 42(7): pp. 1215-1229, 1998.

[33] Fu Y., and Rodin G.J., Fast solution method for three dimensional Stokesian many-particle problems, in *Commun. Numer. Meth. Engng*, vol. 16(2): pp. 145-149, 2000.

[34] Fujimoto, R.M., Parallel and Distribution Simulation Systems, John Wiley & Sons, Inc., New York, NY, 1999.

[35] Govindaraju M., Krishnan S., et al., Merging the CCA Component Model with the OGSI Framework, in *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid (CCGrid2003)*, pp. 182-189, May 2003.

[36] Gorlick, M.M., and Razouk R.R., Using Weaves for Software Construction and Analysis. In *Proceedings of the 13th International Conference on Software Engineering*, pp 23-34, May 1991.

[37] Greengard L., and Rokhlin V., A fast algorithm for particle simulations, in *Journal of Computational Physics*, vol. 73(2): pp. 325-348, 1987.

[38] Greengard L., and Rokhlin V., A new version of the fast multipole method for the Laplace equation on three dimensions, in *Acta Numerica*, vol. 6: pp. 229-270, 1997.

[39] Guyer S.Z., Berger E., and Lin C., Customizing Software Libraries for Performance Portability, *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing*, March 2001.

[40] Guyer S., and Lin C., An Annotation Language for Optimizing Software Libraries, in *Proceedings of the Second Conference on Domain Specific Languages*, pp. 39-53, October 1999.

[41] Hau J., Lee, W., and Newhouse S., The ICENI service adaptation framework, in *Proc. U.K. e-Science All Hands Meeting*, pp. 79-86, 2003.

[42] Hauck F., Becker U., et al., AspectIX an Aspect-Oriented and CORBA-Compliant ORB Architecture, Tech. Report TR-I4-98-08, IMMD IV, Univ. Erlangen-Nürnberg, Sep. 1998.

[43] Hnetynka P., and Plasil F., Dynamic Reconfiguration and Access to Services in Hierarchical Component Models, *Proceedings of CBSE 2006*, pp. 352 - 359, June 2006.

[44] Hoare C.A.R., Communicating Sequential Processes. *Communications of the ACM*, 21(8):666-677, Aug. 1978.

[45] Huang C., Lawlor O., and Kale L.V., Adaptive MPI; in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing* (LCPC 2003), LNCS 2958, pg 306-322, 2003.

[46] Jefferson D.R., Virtual Time, *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404-425, July 1985.

[47] Jifeng H., Josephs M.B., and Hoare C.A.R., A Theory of Synchrony and Asynchrony, *Proceedings of the IFIP Working Conference on Programming Concepts and Methods*, pp. 446-465, 1990.

[48] Kale L.V., and Krishnan S., CHARM++ : A Portable Concurrent Object Oriented System Based On C++, *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, ACM Sigplan Notes, Vol. 28, No. 10, pp. 91-108, 1993.

[49] Kalibera T., and Tuma P., Distributed component system based on architecture description: The SOFA experience, in *Proceedings of Distributed Objects and Applications,* Springer-Verlag, LNCS 2519, 2002.

[50] Kelly W., Roe P., et al., An Enhanced Programming Model for Internet Based Cycle Stealing, in *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 1649-1655, June 2003.

[51] Kohn S., Kumfert G., et al., Divorcing Language Dependencies from a Scientific Software Library. *10th SIAM Conference on Parallel Processing*, Portsmouth, VA. March 12-14, 2001.

[52] Lamport L., Time, clocks, and the ordering of events in a distributed system, *Communications of the ACM*, v.21 n.7, p.558-565, July 1978.

[53] Luckham D.C., Kenney, J.J., et al., Specification and Analysis of System Architecture Using Rapide, *IEEE Trans. Software Eng.*, vol. 21, no. 4, pp. 336-355, Apr. 1995.

[54] Luckham D.C., and Vera J., An Event-Based Architecture Definition Language, *IEEE Operations on Software Engineering*, Vol 21, No 9, pp.717-734. Sep. 1995.

[55] Magee J., Dulay N., et al., Specifying Distributed Software Architectures, *Proc. Fifth European Software Eng*. Conf. (ESEC '95), Sept. 1995.

[56] Magee J., Dulay N., et al., Structuring parallel and distributed programs, *in Software Engineering Journal*, vol. 8(2): pp. 73-82, March 1993.

[57] Magee J. and Kramer J., Dynamic Structure in Software Architectures, *Proc. ACM SIGSOFT '96: Fourth Symp. Foundations of Software Eng.* (FSE4), pp. 3-14, Oct. 1996.

[58] Magee J., Kramer J., and Giannakopoulou D., 1999. Behaviour Analysis of Software Architectures. In *Proceedings of the First Working IFIP Conference on Software Architecture (*Wicsa1), IFIP Conference Proceedings, vol. 140. pp. 35-50, 1999.

[59] Mahmood N., Deng G., and Browne J.C., Compositional Development of Parallel Programs, *Proceedings of the 16th Workshop on Languages and Compilers for Parallel Computing* (LCPC'03), pp. 109-126, College Station, TX, 2-4 October 2003.

[60] Mahmood N., Feng Y., and Browne J.C., A Case Study in Application Family Development by Automated Component Composition: h-p Adaptive Finite Element Codes, *Proceedings of the International Conference on Computational Science* (ICCS'05), pp. 347-354 Atlanta, GA, 22-25 May 2005.

[61] Medvidovic N., Oreizy P., et al., Using Object-Oriented Typing to Support Architectural Design in the C2 Style, *Proc. ACM SIGSOFT '96: Fourth Symp. Foundations Software of Eng*. (FSE4), pp. 24-32, Oct. 1996.

[62] Medvidovic N., Rosenblum D.S., and Taylor R.N., A Language and Environment for Architecture-Based Software Development and Evolution, *Proc. 21st Int'l Conf. Software Eng.* (ICSE '99), pp. 44-53, May 1999.

[63] Medvidovic N., and Taylor R.N., A Classification and Comparison Framework for Software Architecture Description Languages, *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pages 70-93 (January 2000).

[64] Meyer R.A., and Bagrodia R., Path Lookahead: A Data Flow View of PDES Models, *Proceedings of the 13th Workshop on Parallel and Distributed Simulation* (PADS '99), May 1-4, 1999 in Atlanta, Georgia.

[65] Microsoft: Component Object Model Specification 0.9, http://www.microsoft.com, 1995.

[66] Milner R., Parrow J., and Walker D., A calculus of mobile processes (Parts I and II), *Information and Computation*, 100:1-77, 1992.

[67] Mos A., and Murphy J., Performance Management in Component-Oriented Systems using a Model Driven Architecture Approach, *Proc. of IEEE 6th International Enterprise Distributed Object Computing* (EDOC) Conference, pp. 227-237, September 2002.

[68] Mukherjee J., and Varadarajan S., Weaves: A Framework for Reconfigurable Programming, *International Journal of Parallel Programming*, Volume 33, Issue 2 - 3, Jun 2005, Pages 279 - 305.

[69] Newton P., and Browne J.C., 1992. The CODE 2.0 Graphical Parallel Programming Language, in *Proceedings of the ACM International Conference on Supercomputing*.

[70] Object Management Group: Common Object Request Broker: Architecture and Specification, CORBA 2.6.1, formal/02-05-08, ftp://ftp.omg.org/pub/docs/formal/02-05-08.pdf, 2002.

[71] Oden J.T., Diller K.R., et al., Dynamic Data-Driven Finite Element Models for Laser Treatment of Cancer, *Journal for Numerical Methods for Partial Differential Equations*, Accepted for publication, 2007.

[72] Oden J.T., Diller K.R., et al., Development of a computational paradigm for laser treatment of cancer, in *Proceedings of International Conference on Computation Science* (ICCS 2006), pp. 530-537, 2006.

[73] Oldfield R., and Kotz D., Armada: a parallel I/O framework for computational grids, *Future Generation Computer Systems*, vol. 18(4), pp. 501-523, 2002.

[74] Parashar M., and Hariri S., Autonomic Computing: An Overview, *UPP 2004*, Mont Saint-Michel, France, Editors: J.-P. Banâtre et al. LNCS, Springer Verlag, Vol. 3566, pp. 247-259, 2005.

[75] Parashar M., Li Z., et al., Enabling Autonomic Grid Applications: Requirements, Models and Infrastructures, *Self-Star Properties in Complex Information Systems*, Lecture Notes in Computer Science, Springer Verlag. Editors: O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. van Moorsel, and M. van Steen, Vol. 3460, 2005.

[76] Perry D.E., and Wolf A.L., Foundations for the study of software architecture, *ACM SIGSOFT Software Engineering Notes*, volume 17, issue 4, pp 40-52, 1992.

[77] Pervez S., Gopalakrishnan G., et al., Formal verification of programs that use MPI one-sided communication, in *Proceedings of the 13th European PVM/MPI Users' Group Meeting*, LNCS, Springer, pp. 30-39, 2006.

[78] Plasil F., Balek D., and Janecek R., SOFA/DCUP Architecture for Component Trading and Dynamic Updating, *Proceedings of the ICCDS '98*, Annapolis, IEEE Computer Soc. Press, pp. 43-52, 1998.

[79] Plasil F., and Visnovsky S., Behavior Protocols for Software Components, in *IEEE transactions on Software Engineering*, Vol. 28, No. 11, pp 1056-1076, Nov. 2002.

[80] Prakash S., and Bagrodia R., Using Parallel Simulation to Evaluate MPI Programs, *Proceedings of the Winter Simulation Conference*, Washington D.C., Dec. 1998.

[81] Prieto-Diaz R., Domain Analysis: An Introduction. *Software Engineering Notes 15*, 2 , pp: 47-54, April 1990.

[82] Seiter L., Mezini M., et al., Dynamic component gluing, in OOPSLA Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems, November 1999.

[83] Shaw M., DeLine R., et al., Abstractions for Software Architecture and Tools to Support Them, *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 314-335, Apr., 1995.

[84] Siegel S.F., Model Checking Nonblocking MPI Programs, in *Proceedings of 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI 2007, LNCS 4349, pp. 44-58, 2007.

[85] Siegel S.F., and Avrunin G.S., Verification of MPI-Based Software for Scientific Computation. In *Proceedings of the 11th International SPIN Workshop,* (SPIN 2004), LNCS 2989, pp. 286-303, 2004.

[86] Siegel S.F., Mironova A., et al., Using Model Checking with Symbolic Execution to Verify Parallel Numerical Programs, In *Proceedings of the 2006 International Symposium on Software Testing and Analysis* (ISSTA '06) , pp. 157-168, July 17 - 20, 2006.

[87] Sirin E., Hendler J.A., and Parsia B., Semi-automatic composition of web services using semantic descriptions. In *Proc. Workshop on Web Services: Modeling, Architecture and Infrastructure* (WSMAI), pages 17-24. ICEIS Press, 2003.

[88] Stickel M., Waldinger R., et al., Deductive Composition of Astronomical Software from Subroutine Libraries, in *Proc. 12th Intl. Conf. Automated Deduction*, edited by A. Bundy, Springer, 1994, vol. 814 of Lect. Notes Artificial Intelligence, pp. 341--355.

[89] Sun Microsystems: Enterprise JavaBeans Specification 2.0, http://www.microsoft.com, 2002.

[90] Sun X., and Pitsianis N., A Matrix Version of the Fast Multipole Method, in *Siam Review*, vol. 43(2): pp. 289-300, 2001.

[91] Sunderam V., and Kurzyniec D., Lightweight Self-Organizing Frameworks for Metacomputing, in *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing HPDC-11 (HPDC'02)*, pp. 113-124, July 2002.

[92] Swarztrauber P.N., Multiprocessor FFTs, in *Journal of Parallel Computing*, vol. 5: pp. 197-210, 1987.

[93] Szyperski C., Component Software: Beyond Object-Oriented Programming, 2nd edition, Addison-Wesley, Jan 2002.

[94] Taylor I., Shields M., et al., Distributed P2P Computing within Triana: A Galaxy Visualization Test Case, in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS 2003)*, April 2003.

[95] Vetter J.S., and Worley P.H., Asserting Performance Expectations, *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing* (SC'02), pp. 1-13, Balitimore, MD, Nov. 2002.

[96] Wu D., Sirin E., et al., Automatic Web services composition using SHOP2. In *Workshop on Planning for Web Services*, Trento, Italy, June 2003.

[97] Xu K., Takai M., et al., Looking Ahead of Real Time in Hybrid Component Networks, *Proceedings of the 15th Workshop on Parallel and Distributed Simulation* (PADS 2001), May 2001.

[98] Yoon Y., Browne J.C., et al., Productivity and Performance Through Components: The ASCI Sweep3D Application: Research Articles. *Concurrency and Computation: Practice & Experience,* 19, 5 (Apr. 2007), pp. 721-742.

[99] Zhang K., Damevski K., et al., SCIRun2: A CCA Framework for High Performance Computing, *Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments* (HIPS'04), pp. 72-79, 2004.

Vita

Nasim Mahmood was born in Feni, Bangladesh on April 5, 1976. He is the eldest son of Rafique Ahmed and Jahanara Akhter. He has an elder sister Rafiqa Sharmin (Luna) and younger brother Ashique Mahmood (Rupam). He grew up in Dhaka, Bangladesh. He received a B.S. in Computer Science and Engineering from Bangladesh University of Engineering & Technology, Dhaka, Bangladesh in May 2000. He served as a lecturer of Bangladesh University of Engineering & Technology, Dhaka, Bangladesh from July 2000 to August 2001. In August 2001 he entered the Graduate School of The University of Texas at Austin. He received a M.S. in Computer Sciences from the University of Texas at Austin in December 2003.

Nasim has been happily married to his lovely wife, Farhana Wasik (Joya), since 2001. They were blessed with their son, Zafir Abrar Nasim, born on March 19th 2007.

Permanent address: 3373 Lake Austin Blvd, Apt B
Austin, TX 78703 USA

This dissertation was typed by the author.