THE DESIGN OF A GENERAL METHOD FOR CONSTRUCTING COUPLED SCIENTIFIC SIMULATIONS

by

MATTHEW J. SOTTILE

A THESIS

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

June 2001

"The Design of a General Method for Constructing Coupled Scientific Simulations," a thesis prepared by Matthew J. Sottile in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science. This thesis has been approved and accepted by:

Chair of the Examining Committee

6/5/01

Date

Commitee in charge: Dr. Janice E. Cuny, Chair
Dr. Allen D. Malony

Accepted by:

Vice Provost and Dean of the Graduate School

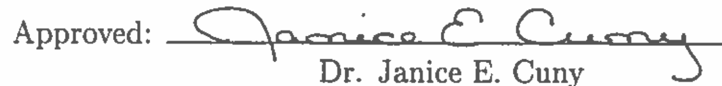
An Abstract of the Thesis of

Matthew J. Sottile for the degree of Master of Science

in the Department of Computer and Information Science

to be taken June 2001

Title: THE DESIGN OF A GENERAL METHOD FOR CONSTRUCTING COUPLED SCIENTIFIC SIMULATIONS

Approved: 

Dr. Janice E. Cuny

With the growth of modern high-performance computing systems, scientists are able to simulate larger and more complex systems. The most straightforward way to do this is to couple existing computational models to create models of larger systems composed of smaller sub-systems. Unfortunately, no general method exists for automating the process of coupling computational models. We present the design of such a method here. Using existing compiler technology, we assume that control-flow analysis can determine the control state of models based on their source code. Scientists can then annotate the control flow graph of a model to identify points at which the model can provide data to or accept data from other models. Couplings are established between two models by establishing bindings between these control flow graphs. Translation of the control flow graph into Petri Nets allows automatic generation of coupling code to implement the couplings.

CURRICULUM VITA

NAME OF THE AUTHOR: Matthew J. Sottile

PLACE OF BIRTH: New Hartford, NY

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon.

DEGREES AWARDED:

Master of Science in Computer Science,
University of Oregon (2001).
Bachelor of Science in Mathematics and Computer Science,
University of Oregon (1999).

AREAS OF SPECIAL INTEREST:

High-performance computing
Functional programming languages

PROFESSIONAL EXPERIENCE:

1999–2001 Graduate Research Assistant, Department of Computer and Information Science, University of Oregon
1997–1999 Undergraduate Research Assistant, Department of Computer and Information Science, University of Oregon
1996–1998 Software Engineer, Blue Cross Blue Shield of Oregon
1994–1996 Software Engineer, InfoLab Technologies, Inc.

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Dr. Janice Cuny, for providing advice and answers to keep me moving in the right direction. I also must thank Dr. Allen Malony for his insightful advice and support on many research projects over the past four years. Thanks to Dr. Michal Young for his help in learning about Darwin and other concepts from the software engineering world, and to Dr. Luciano Baresi for introducing me to Petri Nets. Finally, I'd like to thank Craig Rasmussen and others at the Los Alamos National Laboratory who provided a great deal of valuable feedback regarding my work. This research was supported by a grant from the National Science Foundation, Number ACI-0081487.

DEDICATION

To my family, who never stopped supporting me and encouraging me to pursue my interest in those electronic gizmos we call computers.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION. 1

A process for partially-automated coupling. 3

II. EXISTING WORK. 7

Domain-specific couplers 8
Distributed computing frameworks 9
Other technologies : XML, ADLs 13
Component programming and coupling. 16

III. EXAMPLE MODELS. 19

Laplace equation solver 19
Particle simulation 20
Coupling the heat and particle models 21

IV. DEFINING MODELS AND COUPLINGS. 23

Model descriptions. 25
Defining couplings. 32

V. FORMALIZING MODELS AND COUPLINGS. 34

Formalizing annotated control-flow graphs. 35
Converting a reduced control-flow graph to a Petri Net. 38
Formalizing couplings as Petri Nets 41

VI. RUNTIME SUPPORT. 44

Petri net simulation 45
Runtime system architecture 47

VII. CONCLUSION 51

APPENDIX

A. A BASIC PETRI NET SIMULATOR 53

B. LAPLACE HEAT EQUATION SOLVER 55

C. A BASIC PARTICLE SIMULATION 58

BIBLIOGRAPHY 60

LIST OF FIGURES

Figure Page

1. Transforming descriptions from the simulation to runtime level. 4
2. XML code showing two different contexts using matrices. 14
3. A Darwin component with two input ports and one output port. 15
4. A particle p and the four nearest points on the heat grid h. 21
5. The source code and control flow graph for the iterate function of the Laplace code. 24
6. Two portions of model, each exposing different data sets. 26
7. A function with debugging calls in place. 28
8. Source code for the main body of the particle simulation. 29
9. Annotated control flow graph for the main body of the particle simulation. 30
10. The Darwin interface of the particle model after moving particles. . . . 31
11. Collapsing a while-loop body in an annotated control-flow graph. . . . 37
12. A simple Petri Net with three places and a transition. 39
13. A control-flow graph and the equivalent P/T net. 41
14. Common P/T net synchronization patterns for coupled models. 43
15. Relation of instrumentation to P/T net representation. 46
16. Sequence diagram showing calls made upon reaching a state “s”. 47
17. Runtime architecture overview. 48

CHAPTER I

INTRODUCTION.

The field of computing is driven at some level by the desire to automate complex tasks. Scientific computing - where computers routinely generate solutions to large mathematical models that would be impossible to complete by hand - is a rich source of complex tasks. The immense size of scientific applications has forced the development of increasingly complex high performance computing platforms. Unfortunately, as these computing systems grow, the task of programming and controlling them becomes nearly as complex as the scientific problems themselves. Programming tools are desperately needed to automate common tasks.

In recent years, high performance systems have grown to a point where it is feasible for them to execute more than a single scientific model at a time. Most physical systems are composed of smaller sub-systems which interact to produce effects in the overall system. For example, the ocean is not an isolated system: subtle changes in the atmosphere or sea-ice will change the behavior of the ocean, and similarly the ocean will affect the atmosphere and sea-ice. Scientists working in many different disciplines have already created models of many of isolated sub-systems. They now want to model the more complex interactions between such sub-systems. The most obvious way to do this is not to construct a single, monolithic model of the larger system, but to combine the smaller sub-system models in a manner that mimics the real-world interactions between them. If the scientist has implemented the sub-system

models as computer simulations, a simulation of the larger system can be created by coupling the sub-system simulations. The coupling code then mimics the real-world interactions. Eliminating monolithic simulations of entire physical systems allows tuning of sub-models without unnecessary modification of other sub-models and it makes it possible to reuse sub-models in other simulations.

Unfortunately, coupling computational simulations is often difficult. There are two issues: when is data available for use by a concurrent simulation and what must be done to "translate" that data from one application to the other. Simulations produce a sequence of data during a single execution, and identifying when data is available for external use requires an intimate knowledge of a simulations' internal control state. Due to the complex control flow found in parallel simulations, capturing control state behavior in such a way that it can be used to couple simulations is a very difficult problem. In addition to control state behavior, simulations utilize a wide variety of data structures. Transforming exotic data structures for exchange is a non-trivial problem, especially in environments where speed and space constraints are very restrictive.

We will focus here on control state issues in coupled simulations. A key part of the coupling problem is identifying points during runtime where data can be extracted from or provided to participating simulations. This is a difficult problem from many perspectives. Simulation codes, particularly those that are implemented as parallel or distributed programs, generally have a complex structure. Additionally, couplings are best described in terms of the scientific models the simulations represent, so there must be a mapping between the simulation control states and steps in the solution process for the original scientific model.

Many methods exist for analyzing control structure in complex programs and are implemented in applications for code optimization, debugging, and performance analysis. Unfortunately, no schemes exist for analyzing existing code and describing it in a form suitable for coupling non-trivial programs. Even a complete description of the data and control structure of a simulation is virtually useless in coupling unless control states are mapped to steps in the scientific model. We present a method for constructing a detailed, generic description of the potential couplings of complex codes. We also present a means for transforming them into a form based on Petri Nets that allows specific couplings to be generated and executed within a proposed software runtime architecture. All of these pieces are designed without a specific domain of application in consideration to avoid limiting their applicability.

A process for partially-automated coupling.

We propose a solution to partially automate the coupling of scientific simulation as shown in Figure 1. The figure shows two simulation codes (top left and top right) as the starting point. First, our tools assist the user in creating a model description of each single, stand-alone simulation code that describes its control flow annotated with a set of control points at which couplings can be defined. We will simplify this process through source code analysis and support for user intervention. Next, the user describes an "abstract coupling" which captures exactly how and when data will flow between the simulations in terms of a global state of the coupled simulation, so that a third-party program can monitor and control it at runtime. The coupling descriptions will identify points in the participating models at which data is transferred or control is synchronized. A coupling point will be a pair, with the first element being the set of control states that must be reached for the coupling to occur. The second element

of the pair will be the action that must occur at the coupling point, including data transfer and code that may be executed in the runtime system.

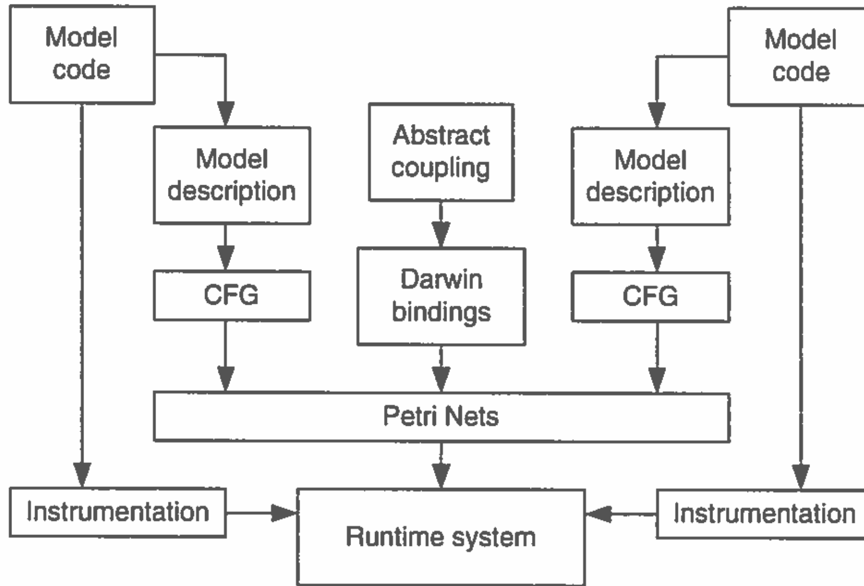


Figure 1: Transforming descriptions from the simulation to runtime level.

Our tools will transform the abstract coupling and model descriptions to a set of annotated control-flow graphs and then to a Petri Net representation suitable for the runtime support environment. To build the Petri Net representation, the control-flow graphs will be extended based on annotations before being converted to equivalent Petri Nets, and applying preconstructed patterns for connecting them will build the global, coupled net. Since actions can be associated with transitions in even basic place/transition (P/T) Petri Nets [15], the actions required at runtime to perform the couplings can be attached to appropriate P/T net transitions. Furthermore, simple extensions on basic Petri Nets (such as colored Petri Nets) allow information to be associated with tokens within the nets, so that coupling-related decisions can be made

in the Petri Net driver portion of the runtime system.

Due to the importance of accuracy and correctness in coupled models, we take a more formal approach to the description and runtime architecture than commonly found in many modern software systems. Formal methods such as Petri Nets and control-flow analysis guarantee some degree of correctness when applied properly. Formalizing coupling as interactions between Petri Nets allows verification that the software will behave in the manner chosen by the user, and Petri Net simulators allow the formal representations to be "executed" in the runtime system.

For reasons of clarity, we introduce terminology that will occur frequently in the remainder of this document. Though most of the terms can be used interchangeably, we must differentiate between them here to avoid ambiguities between references to simulation code and references to the scientific problem being simulated. Terms such as interface, for example, could have multiple meanings: an interface between two scientific "models" would represent something like the surface of the ocean touching the atmosphere, whereas an interface between two scientific "simulations" would be an actual data interface at the code level. A *mathematical* or *scientific model* refers to a mathematical representation of a physical or theoretical process, such as a set of partial differential equations. A *computational model*, also known as a *simulation*, refers to a computer program that approximates a solution for a scientific model.

It is important to note that a simulation of a model can be implemented using a wide variety of techniques. This means that though two models may have the same structure (based on stochastic processes for example), the simulations that solve them may be drastically different. A result of this potential difference in internal structure of simulations is that coupling simulations may not be as straightforward as coupling

mathematical models at the theoretical level. We will assume that the term model refers to a computational model unless explicitly stated.

In the next chapter, we discuss related work. Following that we introduce two simple models that we will use throughout the paper to describe couplings. In Chapter IV, we discuss our model descriptions. In Chapter V, we discuss coupling descriptions in two parts: how the user defines the couplings between models, and how this description is translated into a form usable by a runtime coupling framework. Finally in Chapter VI, we present a runtime framework that allows dynamic coupling of independent models through minimal source code modification and a set of tools for managing the interactions and behavior of each model in the system.

CHAPTER II

EXISTING WORK.

Surprisingly, there is very little in the way of automated tools specifically for the purpose of coupling simulations. The tools that do exist fall into two categories: domain-specific couplers and distributed computing "frameworks." Domain-specific couplers, created for a particular problem domain (such as climate modeling or ecology), support users who are well-versed in their problem domain. Integrating existing, stand-alone simulations within such couplers is not an automated process. The other end of the spectrum are generic programming environments, or "frameworks" for distributed computing. They allow users to glue arbitrary programs together without compiling them into a single executable image. These frameworks are commonly distributed as programming libraries that provide functions for exposing internal data and functionality, and interfaces for addressing data and functionality available in other programs. Unfortunately, the programmer must explicitly define program interactions at the source level, adding instrumentation to interface with the framework. In addition, the framework or environment may impose a style requirement upon the programmer (such as that found in Enterprise Javabeans [11]) which can make integrating existing code into the environment difficult (it may, for example, require a massive overhaul of program structure).

Other types of tools can be applied to the coupling problem to assist users in building coupled simulation environments, although they generally address a small

portion of the task. As we will see later, one of the largest problems in building a generic coupling environment is that of describing both the models and the data passing between them. Technologies for creating "portable" data have emerged recently, in part due to the rising popularity of the Internet as a tool for commerce applications spread over a heterogeneous computing environment. Some formats, such as the extensible markup language (XML), provide not only a format for containing data, but formats for meta-data describing the actual data [12]. Similarly, some members of the software engineering community have been attempting to build languages for describing software itself. Examples of these have existed for many years and are commonly known as architecture description languages (ADLs) [10]. One feature of ADLs that is of interest in coupling is their basis in formal methods. Formal methods provide a means for proving properties required for generic systems that offer scalability, portability, and specialization.

Domain-specific couplers

With the scarcity of generic coupling architectures or tools, users often turn to domain-specific solutions to couple their models. By domain specific, we refer to couplers that are created for a particular problem domain (such as climate modeling or ecology.) A small set of users are targeted, so many assumptions can be made about the couplers' domain of application. Data formats can be restricted if, for example, the coupler assumes that matrices that will be exchanged between models will always be sparse. Assumptions such as this are made for performance gains. Unfortunately, such assumptions become restrictions when applying the coupler to different problem domains. Often different domains require extensive modification to either the coupler or to the models themselves. In other cases, the situation is

even worse: the coupler can not provide functionality that makes sense in the target domain. As an example, features useful for physics simulation, such as computation of temperature gradients, are useless when coupling ecological simulations of species interaction.

One successful domain-specific coupler is used for the Los Alamos Coupled Climate Model [7], and is closely related to work done for the NCAR CSM flux coupler [3]. The LANL climate model is based on a set of four component models: ocean, sea ice, land, and atmosphere. Each component is connected to a central flux coupler, and state variables along the interfaces (such as the ocean/atmosphere boundary) are sent by the components to the coupler. Most work is performed on the finest grid scale over all components. The coupler automatically re-maps each grid to this scale before computing fluxes. Differences in grids and models that cause problems mapping data between models, such as errors representing rivers that connect to the ocean, are resolved by hand. This coupler has proven successful, but is very specific to climate modeling

Distributed computing frameworks

Distributed computing frameworks have existed for quite some time, both as a programming aid to speed up development time and to provide a robust runtime support environment. The motivation behind the development of these environments is similar to that of generic coupling systems: frameworks allow programmers to compose code segments while shielding them from low-level, system interfaces such as RPC or sockets. Because the requirements of the applications that will use these frameworks has some influence in their architecture, we break them into two categories: scientific and enterprise environments. The final two sections of this chapter

will discuss other technologies, particularly those for data and computational interface definition and component programming.

Scientific environments

The scientific computing community has been working on environments for problem solving for well over a decade. Computing platforms reached a point where it was feasible to sacrifice resources to gain abstraction, making the layered "framework" approach possible for building scientific applications. One of the better known, well established systems is the Parallel Virtual Machine (PVM) from Oak Ridge National Laboratory and the University of Tennessee [17]. PVM, MPI [6], and other parallel and distributed programming libraries, provide a layer above the low-level communication mechanism used for moving data and supplies the user with important synchronization mechanisms. Unfortunately, PVM is more more appropriate for building individual models: coupling at the PVM level would involve significant modification of source code to instrument the models with PVM code. A newer project, HARNESS (Heterogeneous Adaptable Reconfigurable NEtworked SystemS) [2], also emphasizes the notion of "virtual machines" but is based on a pluggable model, in which the distributed virtual machines (DVMs) are composed of independent plug-ins.

Users can create plug-ins for common scientific tools, such as BLAS and LAPACK. Thus they can rapidly prototype their programs by reusing existing plug-ins. HARNESS support is dynamic allowing plug-ins to join or leave the system at any time. In addition, DVMs are able to merge and interact, and split back into independent entities at runtime. This is a powerful concept that would be very useful in coupling. If models were implemented to run in their own DVMs, HARNESS pro-

vides the infrastructure for merging these DVMs so that the models can run within a shared computational space. Unfortunately, HARNESS does not provide automated methods for describing these mergings and interactions; it simply provides a substrate on which these sorts of tools could be constructed.

A similar, more mature environment than HARNESS is NetSolve from the University of Tennessee [4]. NetSolve is intended to manage a large, distributed set of resources in a heterogeneous computing environment. The resources may be dynamically introduced and removed from the system based on user requirements or availability of computational resources. Users are able to post requests for work to be performed to the system, and later poll it to retrieve results. Since the system handles assigning work to resources, NetSolve is able to handle issues of load balancing internally without placing this burden on the user. The system comes with tools ready to use for performing linear algebra computations which are common in many scientific applications. The tools are not specific to any particular domain, and can be used by any user for whatever domain they are working in. NetSolve provides fault tolerance, load balancing, and resource management and discovery but it does not provide any explicit facilities for coupling applications. NetSolve, PVM, and related systems are intended for model builders to use in the initial construction of their simulations. They are less suited for coupling existing simulations.

Enterprise environments

In contrast to scientific applications, enterprise applications require very little numerical computation, but a huge amount of data management and manipulation (as found in database systems). In addition, they are designed to support a large number of concurrent users. In the scientific realm, most applications have few (if

any) interactive users. As a result, enterprise programming environments provide mechanisms for managing transactions, such as fault tolerance, sophisticated data locking, and various conflict resolution schemes. Often enterprise systems must be significantly more robust than their scientific computing cousins, because of the high cost of an error. Business application developers do not have the option of hand-picking "responsible, knowledgeable" users, so they must take extreme actions to protect their data: a single corrupt piece of data or loss can result in huge amounts of money lost.

The Common Object Request Broker Architecture (CORBA), a widespread technology adopted by the Object Management Group (OMG) [14], has been largely accepted by the business world for constructing large distributed applications. The basic idea behind CORBA is that objects can be distributed across a wide variety of platforms, implemented in different languages, but connected together using a common intermediate mechanism. Objects are located and attached to with the assistance of Object Request Brokers (ORBs) that contain both a reference to the object itself and an abstract description of its interface. CORBA has been used successfully in taking legacy codes and wrapping them with languages such as C++ so that they can be used without working with outdated technology or huge amounts of poorly documented and written source code. CORBA does have performance issues due to the possibility that applications may run in an "untrusted" environment: ORBs must be implemented securely to prevent intruders from either gaining access to or damaging data.

Another similar, more recent enterprise environment is that provided by the Java programming language, including Enterprise Java Bean and Java RMI tech-

nology [5]. Java tries to solve many problems that add unnecessary complexity to applications written in CORBA by using platform independent bytecode for compiled applications and encouraging strict object-oriented program design. One of the interesting results of Java bytecode and OO techniques is that a program can "discover" the interface of a Java object by examining the bytecode signatures of its public methods and fields. Beans uses this to discover methods that are named using a regular getter/setter interface to access internal data. Unfortunately, this is of little value in coupling because it would require that all models either be written in Java or have Java wrappers. Wrappers require interfacing with non-Java code through native interfaces that are rather difficult to work with, even for experienced programmers. Though Java is slowly shedding its reputation for poor performance, few researchers in the scientific community have pursued it for real simulation applications.

Other technologies : XML, ADLs

As mentioned above, there are many different technologies that do not directly address the coupling problem, but can be used to fulfill some of its requirements. Two important aspects of models that must be exposed during coupling are the data structures that can be imported or exported and the control states at which these actions can occur. The eXtensible Markup Language (XML) [12] has become a popular format for encapsulating data and "meta-data" (that is, descriptions of data formats that follow a common format themselves). Document Type Definitions (DTDs) can be created to describe data formats as well as semantic information beyond simple format interpretation instructions. For example, a DTD could contain tags that allow matrices to be embedded within an XML document. Other tags could be defined for structures such as temperature distributions or magnetic fields, both

of which may involve matrices for their representation. In parsing the pseudo-XML code shown in Figure 2, we can see how the additional tags can provide information about matrices based on the context in which they appear.

```
<TEMPERATURE>
    <MATRIX HEIGHT=N WIDTH=M>
        <ROW>...</ROW>
        <ROW>...</ROW>
        ...
    </MATRIX>
</TEMPERATURE>

<FIELD>
    <MATRIX HEIGHT=K WIDTH=K>
        ...
    </MATRIX>
</FIELD>
```

Figure 2: XML code showing two different contexts using matrices.

This example shows two pieces of data, both represented as a matrix. The matrix tag alone does not allow a program to determine how to interpret the contents of the matrices. By placing the matrices within other tags, contextual information from these tags can give semantic meaning to the matrix. The example shows how a temperature distribution and a field (such as an electromagnetic field) can both be stored as matrices. DTDs allow programs to interpret data. Furthermore, data can carry a DTD with it so that a receiver is not responsible for knowing how all data sources could format their output - the data would be responsible for describing itself.

Describing data only solves part of the coupling problem as we will need information about the control states at which that data can be provided or consumed. Integrating control and data descriptions into a single format would be cumbersome

and difficult. Focusing on control information separately is similar to problems faced by software engineers describing software. One relevant result of their work is the family of architecture description languages, such as Darwin [9] [8] and Wright [1]. Darwin is used as part of the implementation mechanism for our coupler prototype. It is a relatively simple language that allows software to be described as a set of components bound together. The components of the software are independent of each other, only reacting to inputs and providing outputs.

Figure 3: A Darwin component with two input ports and one output port.

A component in Darwin defines a functional unit that can both provide output data to and require input data from other components. A trivial example is shown graphically and textually in Figure 3. The figure defines a component, averager, which takes two floating point numbers and outputs their average. A component that wishes to use the averager will bind two of its outputs to the two input ports (indicated with white circles), and one of its inputs to the output port (indicated with a black circle). Edges in the graphical representation of Darwin illustrate bindings between components. Beyond simply allowing components to be defined, Darwin provides language constructs for defining bindings between them that link input and output ports together. In some ways, this approach mimics data flow models constructed in environments such as LabView and SCIRun [13]. Many users find data

flow modeling to be intuitive and well-suited to rapid prototyping from libraries of existing components. Darwin also supports dynamic interface definitions, an important requirement for describing complex scientific models. Further discussion of Darwin as applied to model coupling in the context of our work will be left for later.

Component programming and coupling.

This rising complexity in computer programs has reached a level of complexity where traditional, monolithic programming approaches have become unmanageable in many domains. A successful strategy in dealing with this complexity is to view applications as being composed of distinct functional and data carrying units known as "components." For example, a simple web browser could be composed of a HTTP interface, HTML parser, and graphical renderer. A programmer can bind separately created components for each of these pieces together to build a web browser, instead of writing them all as a single program. Furthermore, a programmer who wishes to build an HTTP based file transferring agent can reuse the HTTP interface component, and add components that are relevant to file transfer, cutting development time.

A component is similar to an object in modern object-oriented programming languages. Objects provide abstraction at the code level, allowing developers to build their programs from pieces that perform specific duties, contain specific data, and expose only the required interface while hiding implementation details. The object-oriented approach lends itself to code reuse, and has proven successful in cutting development time drastically. Components also provide interfaces, contain data, and hide their implementation. A distinction between the two is that objects refer to the programming language level of development (such as C++, Smalltalk, and Java), while components refer to compiled, executable code. Newer object-oriented systems

such as Java however, blur this distinction because Java objects are represented individually in executable bytecode class files. Ignoring this minor complication, we will use the above definition to distinguishing between objects and components.

Components are an elegant mechanism for hiding some of the complexity in building large, monolithic programs. Scientific models that are composed of a set of coupled sub-models can be viewed at the simulation level as a single system constructed from components that implement each sub-model. The data provided by each component would represent the data within the models at the interfaces between the actual physical systems. Functions provided by the components would be available to allow external programs to manipulate their behavior without manipulating data: this could range from starting and stopping the simulation to forcing the reconfiguration of a grid while the program is running. The basic component model breaks down when considering that the models themselves are independently executing programs with potentially complex control states. Control states can expose or require different data, thus the "components" have dynamically changing interfaces. Most component frameworks do not support dynamic components, which rules out the blind application of component-based techniques to model coupling.

Component-based coupling systems must provide necessary information about each model to effectively connect them at runtime. Existing component systems provide interface definition languages (IDLs) that can be used to define the data structures passing between the models and the functional interfaces that each model provides. These would have to be extended to provide additional information about control states and their effect on the data that is exposed by the component. Furthermore, the descriptions will need to be robust enough to support parallel models,

in which many threads of execution all have independent control states. Each control state could potentially affect the global data state. Naive descriptions might lead to either restrictions on how data could be accessed (such as through a "master" thread), or performance degradation (such as forcing all threads to wait for a common control state to be reached) in situations where alternative approaches could be used.

CHAPTER III

EXAMPLE MODELS.

It will be useful at this point to introduce two simple models that will be referred to throughout the remainder of the document. Each model will represent a very simple physical process, each using different techniques for generating solutions. This allows us to address problems that arise when coupling two simulations that behave in completely different manners. The first model is a simple Laplace heat equation solver that uses Jacobi iteration to approximate solutions to partial differential equations (PDEs). The second is a particle simulation that models interactions of simple particles in a confined two dimensional space. By coupling the models, one can create a very rudimentary model of a thin layer of gas over a heated plate, where the heat model causes changes in the energy of the particles and likewise the particles will transfer some of their energy back to the plate.

Laplace equation solver

A widely used technique for approximating solutions to PDEs involves an iterative process that approximates a solution, relaxes the values in the approximation, and repeats the process until the difference between two iterations converges below some threshold. Different techniques exist for this process, each being best suited to different conditions and having different convergence patterns. The Jacobi iteration was chosen because it is relatively easy to implement and parallelize. The grid used

by the simulation is rectilinear with a fixed size and equally spaced grid points. By nature of the problem, the solution does not involve time in any way: the initial approximation is a "snapshot" of the system at a single point in time and the solution is a refined approximation of that approximation. Intermediate steps represent converging solutions not a heat flow over time. A full implementation of the Laplace solver is presented in Appendix B.

Particle simulation

Particle models are more commonly referred to as N-body problems. Well-known techniques exist for exact solutions to systems composed of 2 or 3 bodies, but for large values of N, approximations must be used. There are many different techniques and the appropriate choice depends on the type of results that users are seeking. In some problem areas, fixed grids can be used that restrict the movement of particles to grid points. We use a less constraining approach that allows particles to move in any direction at any velocity.

We assume that the system is composed of n independent particles and they exist within a bounded two dimensional space measuring w by h units. Each particle has a position (x, y) and a velocity vector (Θ, v). We assume that two particles do not exert any force on each other until they are a small distance from each other, allowing us to consider particles that move only in straight lines along their velocity vector. At any given point in time, the simulation computes the smallest time until the lines of motion of two particles intersect. Once this is found, the simulation advances time to that point, moves all of the particles, and adjusts the motion of those particles that are close enough to interact. Thus each iteration represents a variable size timestep. Further documentation on the model and its implementation

can be found in Appendix C.

Coupling the heat and particle models

Coupling these models presents a few interesting problems, most notably, the Laplace solver does not have a time component while the particle model does, and the Laplace model uses a fixed grid, while the particle model uses a set of points in space. In our coupling, we let the heat model start with the initial conditions of the "plate" over which the particles are placed. After solving the heat equations for a single point in time, the heat distribution and particle motions will be used to compute new particle velocities and changes to the heat distribution on the plate. The particles will be moved, and the process will be repeated. Thus time will be introduced into the heat model because we solve it's equations at each time step of the particle model.

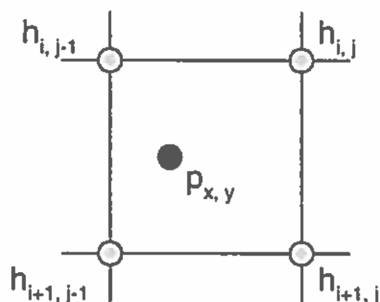


Figure 4: A particle p and the four nearest points on the heat grid h.

When mapping between the particle distribution and the heat grid, particle locations will be approximated based on the constraints of the Laplace model grid. The resolution of the heat model grid dictates a constant distance between any adjacent grid points. A particle is guaranteed to be at most this distance from four grid points,

as we can see in Figure 4. The effect of this particle will be distributed to each grid point based on their distance from the particle. Similarly, the heat input to a particle will be a weighted average of the nearest four grid points.

We will consider the coupling in terms of sequential, single threaded implementations of each model. Each model is easily parallelized and the control flow in the parallel versions is very similar to the sequential versions with the addition of minimal thread communication.

CHAPTER IV

DEFINING MODELS AND COUPLINGS.

In order to automate coupling, the software must have a sufficient description both the models and their desired couplings. The model descriptions must capture enough information about the models (both at the computational and scientific level) so that the software system can generate much of the low-level code for interacting with other models. Model descriptions are independent from coupling descriptions so that a single model description can be reused in multiple coupled contexts with little or no modification. In our work, we use analysis of the simulation codes to assist in generating the model descriptions. We rely more on the user for the coupling descriptions, as they require knowledge at the scientific level that cannot be derived from source code.

The process we use starts with the original, independent simulation codes. We are primarily concerned with capturing sufficient control flow information. Modern compilers already do extensive control flow analysis. Most compilers are structured with a front-end that converts source code into an intermediate, abstract format, and a back-end that generates executable code from this intermediate format. We propose to use the information provided by a compiler front-end along with user input to derive the needed control flow graph of a simulation. For instance, the control flow graph for the `iterate` function of the Laplace example is shown in Figure 5.

Initially we will be concerned with creating the control flow graph for the sim-

```
double iterate(double **a, double **b) {
  double diff, total;
  int i, j;

  total = 0;
  for (i = 1; i < HEIGHT-1; i++) {
    for (j = 1; j < WIDTH-1; j++) {
      b[i][j] = (a[i-1][j] + a[i+1][j] +
                 a[i][j-1] + a[i][j+1])/4;
      diff = a[i][j]-b[i][j];
      total += diff*diff;
    }
  }

  return total;
}
```

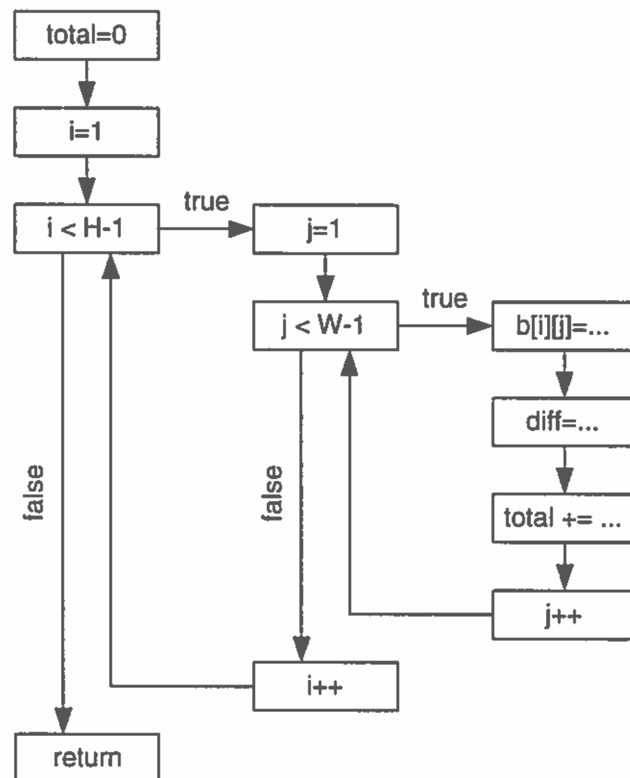


Figure 5: The source code and control flow graph for the iterate function of the Laplace code.

ulation code. Each function contained in the simulation will have an independent control flow graph: embedding the control-flow graph for functions within their caller results in an excessively complex graph, and prevents accurate representations of code behavior such as recursion. Dealing with with functions is simple: nodes within the control flow graph of the function caller will simply point to the control flow graph of the callee. We will allow the user to annotate the control flow graph with points at which data elements start and stop being available to external programs. Based on these annotations we can return to the original simulation code and place instrumentation in appropriate places for creating events associated with data availability changes. We can also use the annotations to generate a static Darwin description of the model interface based on the state it is in at a given time. In Chapter V, we will show how the control flow graph can be converted into a Petri Net for execution in the runtime system.

Model descriptions.

The model descriptions are based on both control flow analysis techniques from compilers and component description methods from the Darwin architecture description language. Existing component description techniques capture data and functional interface information about the models. Unfortunately, in a coupled environment these interfaces may vary dynamically as the internal control state of the models change during execution. The semantics of the exposed data may change as a result. For example, a model may provide data for a specific time scale during one part of a single iteration, and then transform this data for a new time scale during the rest of the iteration. Similarly, the availability of data may also change. For example, if a portion of the code uses iterative methods for approximating solutions to PDEs,

it may expose the rate of convergence while it is iterating. After it has solved the PDEs, it may move on to a different computation where rates of convergence have no meaning, so it will not expose data of that type. This is illustrated in Figure 6, where the value d represents the rate of convergence. A simpler, more common situation is for a model to lock data during critical regions. Within these regions data may not be valid. These types of interface behavior will impact the synchronization rules imposed on coupled models and must be known when gluing them together.

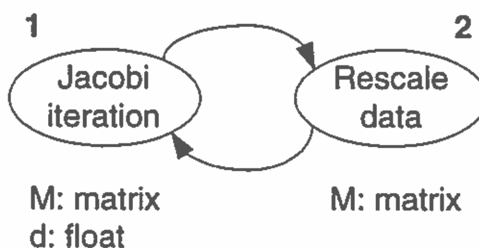


Figure 6: Two portions of model, each exposing different data sets.

To capture the necessary information, our model descriptions will need to specify the set of important control states the model will enter, the data that will be available in these states, and any information restricting access to this data at runtime. We will use finite state machines to represent the control flow of the models. In an ideal environment, we would use the control flow analysis portion of a compiler front-end to derive these finite state machines as control-flow graphs.[1] Though the entire process of describing a model cannot be automated, important and tedious portions can be to some degree. Doing so makes sense to developers as it can both speed up the

[1]For the implementation of basic tools at this time, we have used a simpler, more restrictive method based on static source code analysis and observing runtime behavior of a "typical" run.

process and eliminate errors potentially introduced by hand coding everything. One portion of the description that cannot be easily automated is the choice of data to be exposed and protected, and the protocol by which it should be managed when models are coupled. Our discussion will be based on control-flow information from a compiler front-end.

Identifying important control states: Control flow analysis

To generate a model description, we use the compiler front-end to generate a control flow graph for the simulation. This kind of information is usually available for languages commonly used to build simulations, such as C, C++ and FORTRAN. Not all of this information is relevant for coupling, so we allow the user to filter it. At this point we have a full description of the control states the code passes through for each function within the simulation. Since some of the functions that are included in the simulation are not relevant to coupling, by allowing the user to filter them out of the description we can simplify the description without losing important information. Examples of code that the user may wish to ignore are calls to debugging code or functions to manage internal resources. This information may be useless to other applications participating in a coupled situation, but automated tools cannot determine which functions should be ignored. Consider the code taken from the Laplace equation solver with debugging calls left in place shown in Figure 7.

Analysis of this portion of code would show that after a call to `iterate`, the utility function **debug** is called twice. The user may want to ignore the calls to utility functions since the state in which the Laplace solver is performing Jacobi iteration is of interest. Extra states for utility functions can usually be ignored without changing the meaning of the state machine representing the calling function. Once a model

```
void iterate() {
    debug("BEGINNING ITERATION");
    for (i=1;i<WIDTH;i++)
        for(j=1;j<HEIGHT;j++)
            b[i][j] = (a[i-1][j]+a[i+1][j]+
                          a[i][j+1]+a[i][j-1])/4;
    debug("ENDING ITERATION");
}
```

Figure 7: A function with debugging calls in place.

has been analyzed at the source level and the user has pruned the control flow graph to contain relevant states, an initial finite state representation of the simulation can be produced. Each function has a state machine, and the state machines for each call within are function are not inlined in the state machine of the callers. This prevents problems representing recursive functions.

Dynamic model interfaces

As stated earlier, the interface exposed to other programs by the model is dynamically determined by the current control state. If a "snapshot" of the model is taken at any arbitrary time during execution, a static interface is exposed to the outside. This interface can be described using architecture description languages such as Darwin, capturing the available input and output channels of the model. Before this can be done, however, we must refine the model representations. At any given point in the the control flow graph of a model, a specific set of data is available. Since we do not want to expose all data that is available at any point, we allow the user to annotate the graph with points at which specific data elements become exposed or hidden. Consider the piece of code from our particle model shown in Figure 8. The

particle data, p, is not available until after it has been initialized, and must be protected during the `move_particles` function. Similarly, the timestep, `ts`, is available once it has been computed within the while loop.

```
1.  void main() {
2.      int i;
3.      float ts;
4.      struct particle **p =
5.          make_particle_array(NUMPARTICLES);
6.      for (i=0;i<NUMPARTICLES;i++)
7.          p[i] = make_particle();
8.      while (1) {
9.          ts = compute_timestep(p);
10.         move_particles(p,ts);
11.     }
12.     for (i=0;i<NUMPARTICLES;i++) free(p[i]);
13.     free(p);
14. }
```

Figure 8: Source code for the main body of the particle simulation.

The particle data is initialized on lines 4 through 7. After the for-loop initializing the individual particles is complete, the particle data is available. We only make it unavailable during the `move_particles` call to ensure that the data is consistent: it becomes hidden immediately before line 10 executes and exposed again immediately after it completes. The timestep for the current iteration is not available until it has been computed the first time on line 9. It is available for the entire body of the while-loop after this point and becomes unavailable only after the loop terminates. The control flow graph of this code is shown in Figure 9. Annotations on the edges of the graph indicate the availability of data changing.

At this point we have sufficient information to describe the interface to the

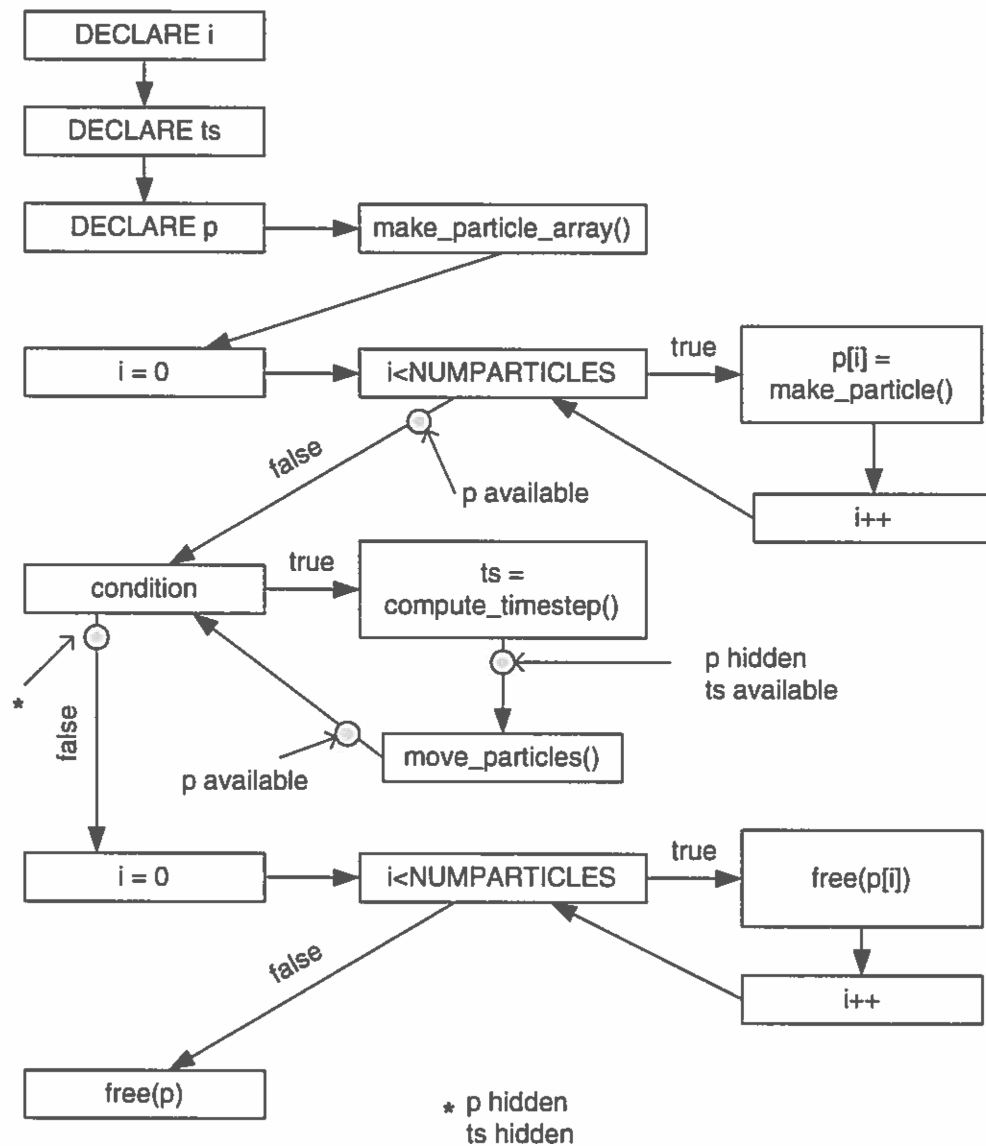


Figure 9: Annotated control flow graph for the main body of the particle simulation.

model using Darwin. The most basic portion of the Darwin language that we will introduce now provides a skeletal description of the model as a component with a set of "provides" ports. These ports represent data that is available. These skeletal descriptions can later be augmented with Darwin code for defining "requires" ports for receiving data and also constraints on bindings between ports. For now, we only have enough information to generate the skeleton. Each skeleton will reflect one particular static interface that the model presents. A sample Darwin interface description is shown in Figure 10 for the interface presented by the particle model immediately after moving the particles in the while-loop.

```
component particle_model {
    provide p<particles>;
    provide ts<float>;
}
```

Figure 10: The Darwin interface of the particle model after moving particles.

For each model, the model description we have built is sufficient to determine the skeleton describing the model interface at any given time by simply examining the current state of the model. When a coupling operation is to occur between two models at a certain state, the models can determine their Darwin interface and exchange the interface information to determine how to connect with the other model. At this point, the model descriptions contain enough information for defining couplings between models.

Defining couplings.

A coupled model is essentially a set of programs with an associated set of "coupling points" at which time data is exchanged or control is synchronized. The process of data exchange involves identifying the source data, destination data, and a transformation that may occur between the source and destination. For example, two models may periodically exchange data on grids of different resolutions. A coupling point for this exchange would identify when the data would be exchanged, and where the data is. In addition, to deal with the differing grid resolutions, a transformation would be necessary to map one resolution to the other. As we can see from this basic example, the coupling definition will require state information about each model to determine which interface the models will be presenting at any given time.

We define a coupling between two models as a point at which time one model stops and requests data from another model. When the requesting model queries the coupler for data, the model holding the data will be checked to see if the data is available. If the data is not available, the requesting model has two choices: it can either block and wait for the data to become available (a blocking coupling) or move on and possibly request the data at a later time (a non-blocking coupling). When the data is available for the requesting model, the coupler must contact the model holding the data and transform it into a format acceptable for the requester. To do so, the data must have type information associated with it at each endpoint. We know that the model providing the data has type information regarding the data because it is part of the Darwin skeletons produced by the model description. The coupling description must include the type of data on the receiving side of the connection.

In order to set up a coupling, we can extend the Darwin interface skeletons

provided by the model description to contain type information on the receiving end of a coupling. The skeleton that is modified is that which is associated with the state of the receiving model at the time a coupling occurs. These points are defined in a similar manner as those in the model description that define where data becomes available. Annotations in the control flow graph of the receiving model will express when and what couplings occur. We use the Darwin "provides" notation to express data that a model exposes, so we can use the opposite notation, "requires," to express the data being received during a coupling. The model providing data does not need to add any information to its interface for the coupling to occur. In addition, we can establish the matching between a providing and requiring port by using the Darwin "bind" command to attach the data ports. We can also easily extend the binding syntax to include the distinction between blocking and non-blocking couplings. Since Darwin is nothing more than a notation for expressing interactions between components, they can be translated by the coupling support system to the appropriate code for establishing connections and transforming data. At this point we now have a full model description based on the control flow of the code, and a set of Darwin bindings that establish the couplings between the models.

CHAPTER V

FORMALIZING MODELS AND COUPLINGS.

In this chapter we will present a method for formalizing the definitions of models and couplings from the previous chapter. Stand-alone models are represented as annotated control-flow graphs, while coupled models are represented as Petri Nets [15]. We use annotations to refine a control-flow graph to formalize model descriptions. We then convert the individual model graphs to Petri Nets, and "glue" them together to build a representation of the coupled system. We do not start immediately with Petri Net representations as most users are less familiar with them and can better understand basic control-flow graphs. The process of gluing Petri Nets together implements bindings between the Darwin component interfaces used to represent couplings.

We apply these methods to a limited variety of parallel models focusing in particular on the SIMD (single instruction stream, multiple data streams) style used in many applications. Threaded programs generally adhere to this model, having many threads of identical code operate concurrently on different, potentially overlapping data elements. The control flow of each thread can be derived by analyzing the single threaded instance of the program. This description could then be duplicated, and the Petri Nets describing each thread can be linked at points where thread executions synchronize. For the purposes of this work, we will only address single threaded models.

The process of formalizing a coupled system as a Petri Net is presented be-

low. First, we transform our model descriptions to integrate control-flow graphs and data related annotations into a larger graph where annotations become nodes in the control-flow graph. Since annotations are used to define the Darwin input and output ports for the model interfaces, we reduce this extended control-flow graph to a form that preserves the control behavior while eliminating many, but not all nodes unrelated to the annotations. Once we have a reduced control-flow graph, we can create a place/transition (P/T) Petri Net representation of the model. Finally, we show how Darwin bindings map to Petri Net "patterns" that can be glued between two model nets to implement the couplings.

Formalizing annotated control-flow graphs.

Our model descriptions include a control-flow graph and annotations on this graph regarding data availability. An annotation within a stand-alone model indicates the data element of interest and whether it is becoming available or hidden at that control point. The nodes of the control flow graph are associated with statements in the original simulation source code. Edges indicate the movement of control through the program as it executes: when a statement completes computation and another statement begins, an edge is placed between the two representative nodes. For statements that cause branching to occur (such as loops and conditionals), an edge connects the statement to the nodes of its possible successor statements. We can now formally define the annotated control-flow graph of a model.

Definition 1 (Annotated control-flow graph) *Let the model be represented as an annotated control-flow graph,* $M = (N, E, A)$. N *is a the set of nodes from the original, unannotated control flow graph.* E *is the set of directed edges connecting*

nodes such that $e = (n_1, n_2) \in E$*, where* $n_1, n_2 \in N$*. The set of annotations,* A*, is composed of pairs* $(e \in E, a)$ *where* a *an annotation on the edge* e*. An annotation* $a = D+$ *indicates a data element,* D*, becoming available. Similarly, an annotation* $a = D-$ *indicates* D *becoming unavailable.*

In order to integrate annotations into the graph as nodes, we must split annotated edges into two edges where a new node representing the annotation can be inserted. We will create a new graph, $M' = (N', E')$, which contains the original graph with annotations as nodes. Without loss of generality, we assume that there is at most one annotation per edge. Creation of M' is as follows:

$$\begin{aligned} \forall(e,a) \in A: \quad & e = (n_1, n_2) \; ; \; a = \{D+, D-\} \\ & N' = N \cup \{n_a = \text{ node labeled with } a\} \\ & E' = (E - \{e\}) \cup \{(n_1, n_a), (n_a, n_2)\} \end{aligned}$$

Finally, we can reduce this graph to a new graph M_R, which eliminates as many non-annotation related nodes as possible. Each node in M' is either labeled with an annotation or not. The reduction algorithm works by identifying sequences of non-annotated nodes and collapsing them down into single nodes. Edges leading into the collapsed node correspond to entry into the sequence, and the edges leaving the node correspond to edges leaving the sequence. A full reduction algorithm capable of reducing loops and branching constructs would be unnecessarily complex for this discussion. The purpose of reduction is primarily to minimize the number of events generated by the models as they execute. In order for a coupler to track the state of the model, an event must be generated corresponding to each node in the control-flow graph. An un-reduced control-flow graph would create an event for nearly every

operation in the program, the majority of which are useless to the coupling system.

Our simplified algorithm searches for sequences of nodes that have single edges leading out from them, thus preventing it from attempting to reduce branches. This can be done with a simple depth-first traversal of the graph. If a node has two edges leading from it, we will leave it in the graph. For realistic simulations, we expect many sequences of statements that will have corresponding nodes without annotations. Data availability will be decided before the sequence starts and after it ends. For example, if a loop computes the contents of a matrix with a sequence of operations and requires the data to be locked during computation, the data can be marked as unavailable before entry into the loop and marked available again after the loop completes. The body of the loop can be collapsed to a single node. We show an example of this in Figure 11, indicating annotated nodes with a grey shading.

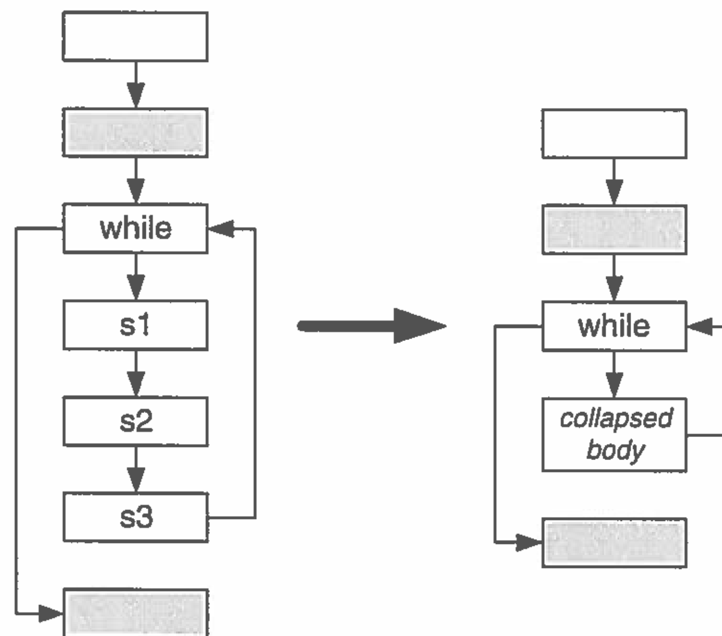


Figure 11: Collapsing a while-loop body in an annotated control-flow graph.

When a sequence of nodes $(n_i, \cdots, n_j)$ is collapsed to a new node n_R, we must adjust all relevant edges leading to these nodes so that each edge $e = (n, n_k)$, with $i \leq k \leq j$, is replaced with a new edge $e = (n, n_R)$. All edges within the sequence, where $e = (n_p, n_q)$ with $i \leq p, q \leq j$, are eliminated from the edge set. Finally, the edge $e = (n_j, n)$ indicating the edge leaving the sequence must be replaced with a new edge $e = (n_R, n)$. Since events are associated with control-state changes, we want to cut down on the number of useless events that are generated. Collapsing a sequence of k nodes will reduce the number of events generated for the corresponding control-states to the number of events generated by a single node. Given this reduced control-graph, we will be able to correlate nodes with locations in the source code and insert instrumentation to generate events for the runtime system.

Converting a reduced control-flow graph to a Petri Net.

So far we have only dealt with stand-alone models with annotations that expose data. Unfortunately, we cannot easily use the reduced control-flow graph to express couplings. A final transformation must now be applied to the graphs to put them in a form usable for expressing couplings. Petri Nets meet all of our requirements for representing couplings: they are able to express concurrent execution and synchronization between concurrent nets. We will use the simplest form of a Petri Net - a place/transition net, or P/T net. A P/T net is a bipartite, directed graph composed of nodes known as places, and nodes known as transitions. By requiring the graph to be bipartite, we only allow edges (n_i, n_j) where if n_i is a place, n_j must be a transition. Similarly, if n_i is a transition, then n_j must be a place. For any transition t, the *preset* of t is the set of places $n_{pre} \in preset$ such that an edge (n_{pre}, t) exists. Similarly, the *postset* of t is defined to be the set of places $n_{post} \in postset$ such that

an edge (t, n_{post}) exists. Each place is marked with zero or more tokens. A transition "fires" when each place in its preset contains at least one token. When the transition fires, one token is removed from each place in its preset and one token is added to every place in its postset. A simple unmarked Petri Net is shown in Figure 12.

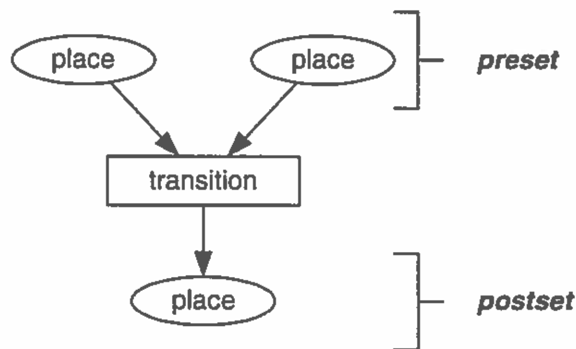


Figure 12: A simple Petri Net with three places and a transition.

Definition 2 (Place/Transition Net) *Let the net $N = (P, T)$, where P is a set of places, and T is a set of transitions. Each transition $t \in T$ is a pair $(pre, post)$, where pre is the preset and post is the postset of the transition. The preset and postset are composed of places $p \in P$. Every place $p \in P$ has a counter associated with it representing the number of tokens in that place.*

To map the reduced control-flow graph M_R to a P/T net, we must create places and transitions corresponding to each node in the graph. Edges in the graph are represented by populating the presets and postsets of transitions with the proper places. To start, let T be populated with one transition for every node in the graph. For each edge $e = (n_i, n_j)$ in M_R, we will look at the transition $t_i \in T$ associated with n_i, and $t_j \in T$ associated with n_j. If there is a place $p \in P$ in the preset of t_j, then

we will add this place to the postset of t_i. Otherwise, we will create a new place in P and add it to the preset of t_j and the postset of t_i. Once we have done this for all edges in the control-flow graph, we must add one new place to the preset of each transition, and add these places to the place set. Similarly, for any transition with an empty postset, we create a single new place and add this place to each empty postset. Places added to the presets that do not occur in any postset correspond to events in the control-flow graph.

Definition 3 (Mapping control-flow graphs to P/T nets) *The mapping between control-flow graphs and place/transition nets starts with a net $N = (P, T)$ where both P and T are empty. Performing the following steps in order will fill the place and transition sets based on the contents of the reduced control-flow graph $M_R = (N, E)$.*

1. *$\forall n \in N$, add transition t_n to T.*

2. *$\forall e = (n_1, n_2) \in E$, if $preset_{t_{n_2}} = \{\}$ then add a place p_{n_2} to P, set $preset_{t_{n_2}} = \{p_{n_2}\}$ and set $postset_{t_{n_1}} = postset_{t_{n_1}} \cup \{p_{n_2}\}$. Otherwise let $p_{t_{n_2}} \in preset_{t_{n_2}}$, and set $postset_{t_{n_1}} = postset_{t_{n_1}} \cup \{p_{t_{n_2}}\}$.*

3. *$\forall t \in T$, add a place p_t to P and set $preset_t = preset_t \cup \{p_t\}$.*

4. *Add a single place p_{end} to P. $\forall t \in T$ with $postset_t = \{\}$, set $postset_t = \{p_{end}\}$.*

When an event is generated, a marking is made in the appropriate place. This forces the net to wait until the event actually occurs before firing the transition corresponding to the event generating node in the control-flow graph. The single shared place that is added to all empty postsets corresponds to the end of the program or function represented by the control-flow graph. Figure 13 shows a control-flow

graph and the equivalent P/T net. Tokens and labels indicating the correspondence between nodes and P/T net parts are not shown, as the figure would become too cluttered.

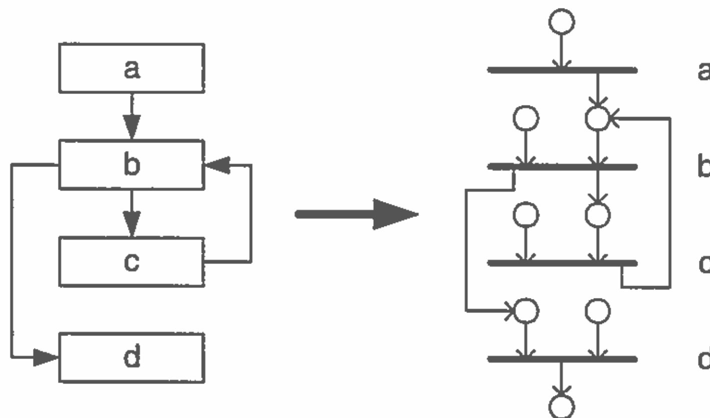


Figure 13: A control-flow graph and the equivalent P/T net.

Formalizing couplings as Petri Nets

The process of formalizing couplings as Petri Nets is simple in comparison with the transformations required to map model descriptions to nets. As stated in the previous chapter, couplings are defined as bindings between Darwin ports in the models to couple. The receiving model represents a location at which a coupling occurs as an annotation similar to those used to expose data in the control-flow graph. We can integrate these annotations into the graph with an identical method. Once these annotations are part of the graph, and we create the P/T net associated with the model, we can glue small P/T nets into the model net to represent interactions between the model and the coupler.

When a coupling point is reached and the coupling occurs, the transition in the

model associated with the annotated receiving node will not be allowed to fire until the coupler notifies the model that the data is received. We add a place to the preset of the receiving transition, and connect it to the postset of the transition associated with the coupler requesting data. This transition then has two places in its preset: one that is not in the postset of any transitions, and one that is added to the postset of the transition associated with the state immediately before the receiving transition. This allows the net to stop and wait for the coupler to respond. The place that was added to the coupling transition that does not occur in any postsets is marked when the coupler either finishes receiving the data or immediately if it is a non-blocking coupling point and the data is not available. A table of very common patterns for different types of synchronization and coupling are shown in Figure 14. We indicate places that are part of the coupling patterns as grey nodes, while the model nets are indicated in white. Extending these patterns to more than two models is trivial.

At this point, we have described the mapping of model descriptions into Petri Nets, and the manner through which nets related to coupling points are attached. The nets themselves are not executed, but they are used to capture and generate event information for the runtime environment to use. We also see that by using places in presets that are not in the postset of any transition, we can force the Petri Net to not fire transitions until the runtime system marks these places. This allows the runtime system to ensure that synchronization protocols are implemented properly based on the semantics of couplings defined by users. In the following chapter we outline a basic runtime system to support coupling using the formalisms we have presented. Further details regarding the implementation of our Petri Net simulator are found in Appendix A.

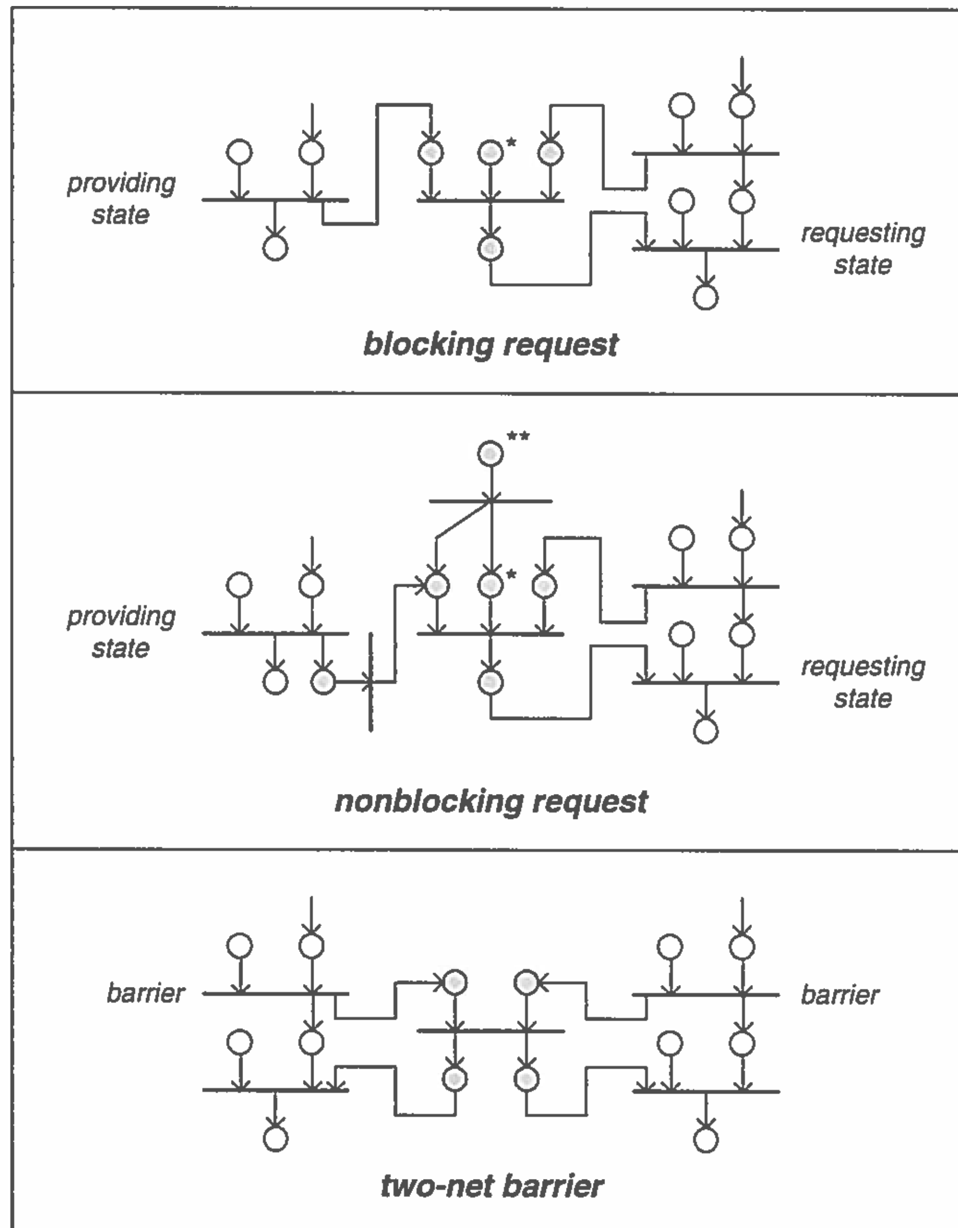


Figure 14: Common P/T net synchronization patterns for coupled models.

CHAPTER VI

RUNTIME SUPPORT.

Implementation of the formal definition of models and couplings is the most valuable result from our work for scientific users. Up to this point, we have only described the theoretical aspects of coupling using Petri Nets and control-flow graphs. In this section, we discuss how Petri Net representations of both models and couplings can be loaded into a P/T net simulation engine and tied to the models via instrumentation and transition callbacks. Instrumentation inside models will notify the coupling system of internal control state changes. When these changes occur, new markings are made within the Petri Nets and the simulator fires all ready transitions. Code attached to coupling net transitions allows the system to execute communication and synchronization code to appropriately exchange and transform data. The separation of coupling code from the model code allows us to centralize the coupling logic while minimizing the amount of new code required within the models themselves.

Before proceeding, we should consider the effects of compiler optimizations on instrumentation and the bindings of the Petri Net simulator to the running code. The instrumentation relies on the ordering of statements matching those expected from the original code analysis. Unfortunately, many optimizers rearrange code to make the best use of pipelined architectures and a variety of cache disciplines. This reordering could render the model description useless, as it may not match the unoptimized original source code. This problem arises in other systems such as TAU, where

instrumentation is the basis for making performance measurements [16]. Efforts are underway to make "instrumentation aware" compilers, but until this takes place in a production development environment we will have to assume that any optimizations made to the model codes do not have side effects in instrumentation. For now, it will be safest to assume that we will be working with unoptimized code.

Petri net simulation

As we showed previously, a control state can be represented as a Petri Net where the parts of the state (entry, computation, and exit) can be represented edges between places and transitions. When a state is reached, instrumentation in the code notifies the Petri Net simulator and a marking is made in the place associated with that state. Since transitions automatically mark their postset, to prevent the entire net from firing independently of the models themselves, each transition associated with a model state has an additional place that occurs in no transition postsets. These places were added when we created the Petri net from our control-flow graphs in the previous chapter. This relation of instrumentation to place marking is illustrated in Figure 15.

Petri net simulation also allows us to associate events with the firing of transitions. In particular, we may want some action to occur outside the model such as data registration or resource allocation. Thus the order of operations for firing a transition would be to first wait for the preset to be filled, execute the function associated with the transition, and fill in the postset when this function completes execution. A sequence diagram shown in Figure 16 illustrates the set of calls made from the model down into the Petri net simulator. The function called upon firing the transition associated with the marked state is labeled as the coupling callback. As we

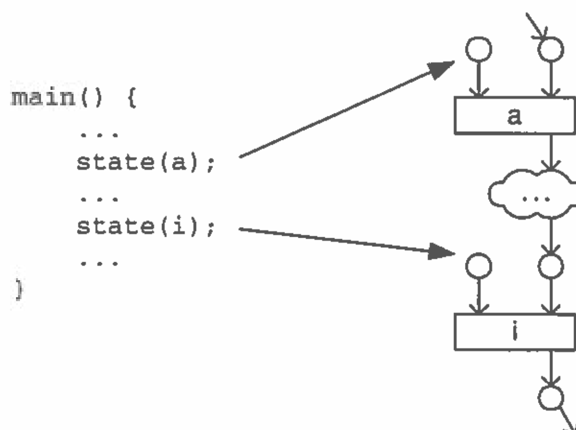


Figure 15: Relation of instrumentation to P/T net representation.

see in Figure 14, the coupling patterns that are added to the net have places that are filled by the coupler. In the case where a non-blocking request occurs, we have two places that are marked by the coupler depending on the outcome of the request. If the data is available, the coupler marks the indicated place for the pattern transition with three places in its preset. If the data is not available, then this preset would not receive a token from the model providing data, so an additional place is available for the coupler to mark. Marking this place fires a transition that forces markings into two of the places in the preset of the waiting transition, thus forcing it to fire.

When the Petri net simulator is initialized, the nets representing the models will be loaded in addition to nets associated with couplings. The transformations described in the previous chapter are performed outside the simulator. The final nets are passed to the simulator for execution. A callback interface is used to bind the simulator to the rest of the runtime system, allowing code associated with the firing of transitions to be executed.

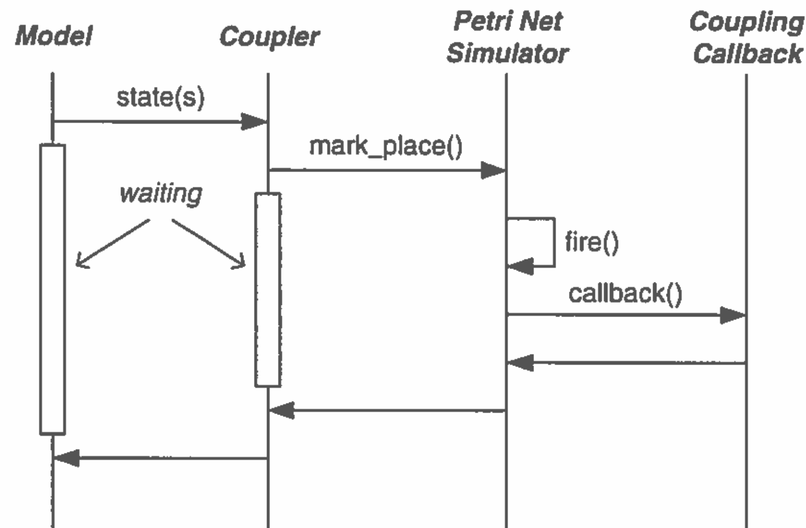


Figure 16: Sequence diagram showing calls made upon reaching a state "s".

Runtime system architecture

The runtime system is based on two important pieces: libraries attached to the individual models and a centralized management server as shown in Figure 17. The libraries provide a means for bridging between the models and runtime system. Instrumentation placed in model code makes calls into the coupling library for operations such as detecting state changes, data registration, and various utility functions. This library then communicates with the central management server to track the individual models and take control during couplings to ensure that they behave as expected.

Model coupler libraries

The coupling library provides the code required on the model end connect to the central management services and communicate directly with other models. Instru-

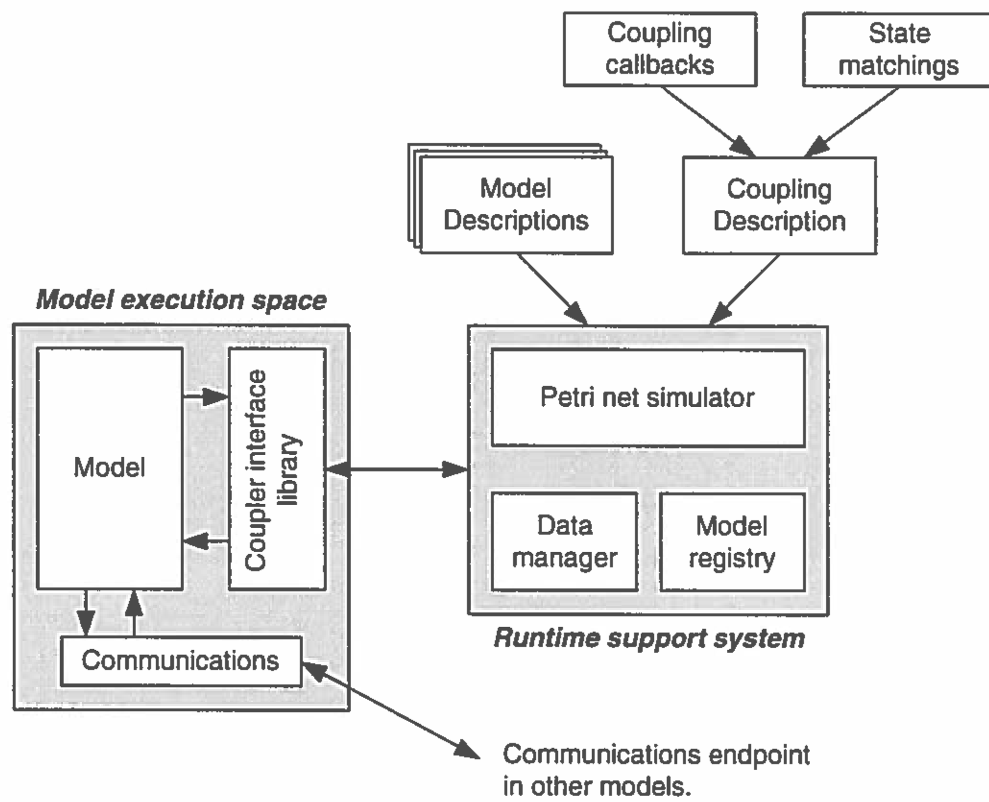


Figure 17: Runtime architecture overview.

mentation within the models provides the necessary calls to the library to send events and requests to the central server. Since the control-flow graphs were generated by the compiler, they contain information regarding the source code location associated with the graph nodes. We can use this information to determine where instrumentation needs to be placed so that state change events are generated at the proper places. Instrumentation to track state changes is very simple. We can add calls to a function "`state()`", where the argument is the state that the model has entered. These state identifiers will be associated with states in the control-flow graph, and therefore they will map to places to mark in the Petri net. The coupler library will send the event associated with state entry to the Petri Net simulator, and the simulator will mark the appropriate place. During the instrumentation call, the thread of execution it is contained within will block until the call returns. If the event being marked in the Petri Net is associated with data becoming available, the system may block the model if another model is waiting for the data. When the data has been transferred, the instrumentation call will return and the model will continue. In other cases, we can send the event to the Petri Net simulator and immediately return control to the model.

Central resource management and tracking

The second portion of the coupling runtime system involves a centralized server that maintains information pertaining to the global state of the coupled environment. This includes not only the Petri Net simulator for tracking control state, but data registration facilities for allowing data to be located and described for conversion purposes at coupling time. Since we do not focus on data issues, we will simply assume that the data involved in this initial prototype will be very simple and requires little

description. Since each model has a set of Darwin component definitions associated with it to capture the changing interface it exposes, the management server must have a list of all types that are bound to the Darwin ports. For each type, the server can maintain a list of other types that are equivalent, or a list of converters to other types that are available. When a binding between two ports occurs, the management system can simply look up the types of the endpoints to determine how data must be manipulated in order to flow between the models. Any transformation code required can be invoked when the coupling system sets up the communication line between the two models.

CHAPTER VII

CONCLUSION

It is an accepted fact that with current programming techniques and computing systems, full automation of all processes cannot be achieved. Simulations are only approximate representations of scientific phenomena and from their implementation source code and runtime behavior one cannot derive their meaning in the theoretical domain. Fortunately, working with these simulations need not be restricted to tedious source level analysis and modification by humans. If automated tools are created for computable tasks, user intervention to aid these tools can provide required information for further automation. Coupling is a tedious task in many ways, and scientific users should be able to work on the aspects of it that they know best, which are those that fall within their domain of expertise. We have shown that many of the activities common to most coupled systems that involve linking programs at the computational level can be performed by the computers themselves. The scientists will no longer have to wear both the hat of the scientific researcher and the computer scientist, but will be forced to work with the computing side of things as minimally as possible.

The steps to implementing computational solutions to problems are fairly well known and restated throughout computer science. First, the problem must be defined so that it is clear and unambiguous, as computers do not implement ambiguous, nondeterministic solutions well at the current time. Given a well defined problem, the problem can be formalized in such a way that generic techniques can be applied to

both provide solutions and allow some level of verification and analysis. Additionally, formalism then allows users to approach implementation using tools that have been previously built, ranging from existing software to algorithms that simply need to be implemented based on their definition. We have given a rigorous definition of what it means to couple two programs together, keeping in mind that these are not generic programs, but scientific simulations. Given these definitions, we have shown that they can be formalized using basic automata theory and mapped into a similar but more appropriate formalism using Petri Nets. Years of research have led to software support and analysis techniques for dealing with Petri Nets, particularly in the area of concurrent computing. By defining and formalizing coupled models as Petri Nets, we can use this existing body of work to build tools to automate a significant portion of the implementation process for coupling. Doing so will save time and reduce errors introduced into hand-coded couplers, allowing scientists to focus on their work and not tedious programming tasks.

APPENDIX A

A BASIC PETRI NET SIMULATOR

The core of the coupling runtime system is a Petri Net simulator that can maintain information about the global control state across all coupled models. For the prototype runtime system we have implemented we have used a simple simulator that is written in Java. It is composed of a core place/transition (P/T) net simulator, a parser for loading human readable Petri Net descriptions, and a set of Java RMI classes that provide access to the simulator interface from external programs. A Petri Net object is composed of place and transition objects, with edges between places and transitions represented as object references. Each transition has a single callback object associated with it, which must be either a null reference representing no callback, or an object implementing the callback interface provided.

For the purposes of our coupling system, we create a single `PetriNet` object for each call that is contained in each model description. When calls are made, the caller places a token in the appropriate place to start the called function. When the call exits, the token produced by the transition associate with the exit is placed in the preset of the transition directly following the call in the caller. To represent this in the simulator, we note that each `PetriNet` object is completely independent from the others, including the `Place` and `Transition` objects inside each. To bind them together, references to `Place` objects at the linkings between the nets are exchanged, so that the preset of a transition in a function will include a reference to a place in

the caller net. `PetriNet` objects are also provided with a label that is used to identify the context in which a transition callback occurs.

Currently, transition callbacks are very simple and are not provided much information about their context by the simulator itself. In the future, the simulator could be extended to allow information carrying tokens which are given to the callbacks, letting the net pass information around based on the current and previous states of the system. All callback objects must implement the `Callback` interface, which defines a function named `callback()` which takes as its parameter the label of the `PetriNet` containing the transition it is bound to. It is up to the callback object implementor to use this label information to determine what actions must take place. Since we will be using a small set of synchronization patterns to bind model Petri Nets together, these patterns can be named appropriately to indicate what coupling point they represent. Based on this information, the callback can then look up in the coupling description what actions must occur. Performing these actions involves making the appropriate calls into the coupler runtime system to move and transform data. The actual communication, transformation, and other data related operations are performed elsewhere and are only triggered by the Petri Net simulator.

APPENDIX B

LAPLACE HEAT EQUATION SOLVER

The following is the source code for a C version of the Laplace heat equation solver. Output and input code has been eliminated to unclutter the basic Jacobi iteration algorithm. This version will initialize the 2D plate to a temperature of zero with one side of the plate set to 100 degrees. After relaxation, the temperature distribution will reflect the diffusion of heat into the plate from the heated side.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define WIDTH     200
4 #define HEIGHT    200
5 #define THRESHOLD 1.0
6 /**
7  * initialize with initial conditions
8  */
9 void init(double **m) {
10   int i, j;
11   for (i = 0; i < HEIGHT; i++)
12     for (j = 0; j < WIDTH; j++)
13       if (i != 0) m[i][j] = 0;
14       else m[i][j] = 100;
15 }
16 /**
17  * iterate
18  */
19 double iterate(double **a, double **b) {
20   double diff, total;
21   int i, j;
22   total = 0;
23   for (i = 1; i < HEIGHT-1; i++) {
```

```
24      for (j = 1; j < WIDTH-1; j++) {
25        b[i][j] = (a[i-1][j] + a[i+1][j] +
26       a[i][j-1] + a[i][j+1])/4;
27        diff = a[i][j]-b[i][j];
28        total += diff*diff;
29      }
30    }
31    return total;
32  }
33  /**
34   * Main program
35   */
36  int main() {
37    double **a, **b, **c;
38    double *aa, *bb;
39    double d1, d2;
40    int area, i;
41    area = WIDTH*HEIGHT;
42    /* allocate arrays */
43    aa = (double *)malloc(area*sizeof(double));
44    bb = (double *)malloc(area*sizeof(double));
45    a = (double **)malloc(HEIGHT * sizeof(double *));
46    b = (double **)malloc(HEIGHT * sizeof(double *));
47    for (i = 0; i < HEIGHT; i++) {
48      a[i] = aa + WIDTH*i;
49      b[i] = bb + WIDTH*i;
50    }
51    /********************************
52     * MAIN BODY OF CODE
53     ********************************/
54    init(a); /* initialize matrix */
55    d1 = 0; d2 = 1000;
56    while (1==1) {
57      d1 = iterate(a,b);
58      if (abs(d1-d2) < THRESHOLD) break;
59      else d2 = d1;
60      /* swap array pointers */
61      c = a; a = b; b = c;
62    }
63    /**
64     * clean up
```

```
65    */
66   free(a); free(b);
67   free(aa); free(bb);
68   return 0;
69 }
```

APPENDIX C

A BASIC PARTICLE SIMULATION

The particle simulation we wrote to couple with the Laplace heat equation solver is a basic implementation of an N-body simulator for the "billiard-ball problem." This problem involves dealing with particles that are allowed to move without being constrained to a fixed grid. We constrain the particles to a fixed size 2-dimensional space, and populate it with particles. Each particle has a position within this space and a vector representing the magnitude and direction of its motion. The magnitude of this vector, the scalar velocity of the particle, is allowed to take on any value within a given range of floating point numbers. Similarly, the direction of motion is allowed to be any floating point value between 0 and 2π.

The fundamental issue in solving the billiard-ball problem is determining the amount of time a single timestep covers when advancing the simulation. If a fixed timestep is chosen, collisions may be missed that occur during that time step. The particles in this case could move "through" each other without interacting. Similarly, if the timestep is chosen to be too fine, many iterations could be spent with particles moving but not interacting. We can avoid these problems by computing the timestep as the time to the next possible collision from a given state. This is computed by calculating all possible intersections between the lines of motion for each pair of particles, and finding the minimum time to these collisions.

Particles are represented within the simulation as pairs, with one element repre-

senting a point in space and the other element representing the motion vector of the particle. When coupling to a simulation such as the Laplace heat equation solver, we must translate these positions to positions on the heat distribution grid. Similarly, we must transform the motion of the particles into a value representing the energy of the particle so that we can transfer this energy to the heat distribution in the Laplace model. In the opposite direction, heat input from the Laplace model to the particles must be transformed into changes in particle velocities. The net effect should result in a speed up of slow particles over "hot" areas of the plate, where the plate will cool down slightly as the particle takes away some of its energy. The opposite case also may occur, where a fast moving particle will lose energy and slow down as it passes over cool areas of the plate, causing the portion of the plate it passes over to heat up.

BIBLIOGRAPHY

[1] Robert J. Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon University, May 1997.

[2] M. Beck, J. Dongarra, G. Fagg, G. Geist, P. Gray, J. Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulous, S. Scott, and V. Sunderam. Harness: A next generation distributed virtual machine. In *Submitted to International Journal on Future Generation Computer Systems*, 1999.

[3] Frank O. Bryan, Brain G. Kauffman, William G. Large, and Peter R. Gent. The NCAR CSM flux coupler. Technical Report TN-424+STR, National Center for Atmospheric Research, May 1996.

[4] Henri Casanova and Jack Dongarra. Netsolve: A network server for solving computational science problems. In *Proceedings of SC1996: International Conference on High Performance Computing and Communications*, 1996.

[5] Jim Farley and Mike Loukides. *Java Distributed Computing.* O'Reilly and Associates, 1998.

[6] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using* MPI, *2nd Edition.* MIT Press, 1999.

[7] Philip W. Jones, Robert C. Malone, and C. Aaron Lai. The los alamos coupled climate model. In *Proceedings of the Second International Workshop on Software Engineering and Code Design in Parallel Meteorological and Oceanographic Applications*, 1998.

[8] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proceedings of Fifth European Software Engineering Conference*, 1994.

[9] Jeff Magee, Naranker Dulay, and Jeff Kramer. Structuring parallel and distributed programs. In *Proceedings of International Workshop on Configurable Distributed Systems*, 1993.

[10] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of the Sixth European Software Engineering Conference*, 1997.

[11] Richard Monson-Haefel. *Enterprise Javabeans.* O'Reilly and Associates, 2000.

[12] Simon North and Paul Hermans. *Teach Yourself XML.* Sams, 1999.

[13] Steven G. Parker and Christopher R. Johnson. Scirun: A scientific programming environment for computational steering. In *Proceedings of SC1995*, 1995.

[14] Alan Pope. *The* CORBA *Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison-Wesley, 1998.

[15] Wolfgang Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*. Springer, 1998.

[16] Sameer Shende. *The Role of Instrumentation and Mapping in Performance Measurement*. PhD thesis, University of Oregon, August 2001.

[17] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, pages 315–339, December 1990.