SUPPORT FOR MODEL COUPLING: AN INTERFACE-BASED APPROACH

by

THOMAS FRANCIS BULATEWICZ, JR.

A DISSERTATION

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
Doctor of Philosophy

June 2006

"Support for Model Coupling: An Interface-based Approach," a dissertation prepared by Thomas Francis Bulatewicz, Jr. in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:

Dr. Janice E. Cuny, Chair of the Examining Committee

May 23, 2006
Date

Committee in Charge: Dr. Janice E. Cuny, Chair
Dr. John Conery
Dr. Michal Young
Dr. Pat Bartlein

Accepted by:

Dean of the Graduate School

An Abstract of the Dissertation of

Thomas Francis Bulatewicz, Jr. for the degree of Doctor of Philosophy

in the Department of Computer and Information Science

to be taken June 2006

Title: SUPPORT FOR MODEL COUPLING: AN INTERFACE-BASED APPROACH

Approved: _____

Dr. Janice E. Cuny

There is an increasing need in the scientific community for the comprehensive simulation of complex, dynamic, physical systems. Often such simulations are built through model coupling, that is, the merging of existing, component models so that their concurrent simulations affect each other. Model coupling is, however, a nontrivial task that is not adequately supported by existing frameworks which often require direct manipulation of model source code. This work presents an approach to model coupling that avoids this hurdle, allowing for the fast-prototyping of coupled models.

Our approach to model coupling allows the scientist to work with a novel model representation, called the Potential Coupling Interface (PCI). The PCI is an

abstraction that exposes only those aspects of a model relevant to coupling, and it is the basis for specifying couplings. Specifically, this dissertation contributes

- the design of a new representation, the PCI, for model coupling interfaces,
- the design of a domain-specific language, called the Coupling Description Language, for describing the coupling of models in terms of their PCIs, and
- the implementation of a prototype coupling environment for Hydrological models.

We conclude that the use of the PCI model interfaces makes it possible to quickly design and execute prototypes of model couplings for further experimentation and investigation. This dissertation research aims to influence the way in which model coupling is practiced in the scientific community.

CURRICULUM VITAE

NAME OF AUTHOR: Thomas F. Bulatewicz, Jr.

PLACE OF BIRTH: Philadelphia, PA

DATE OF BIRTH: February 6, 1978

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
University of Rochester

DEGREES AWARDED:

Doctor of Philosophy in Computer and Information Science, 2006, University of Oregon
Master of Science in Computer and Information Science, 2003, University of Oregon
Bachelor of Science in Computer Science, 2001, University of Rochester
Bachelor of Arts in Religion, 2001, University of Rochester

AREAS OF SPECIAL INTEREST:

Distributed Systems
Modeling and Simulation
Domain-specific Environments

PUBLICATIONS:

Bulatewicz, T. and J. Cuny. 2005. Interface-based support for model coupling: Spatial representation and compatibility issues. Proceedings of the 8th International Conference on GeoComputation. Ann Arbor, MI.

Bulatewicz, T., J. Cuny, and M. Warman. 2004. The potential coupling interface: Metadata for model coupling. Proceedings of the 2004 Winter Simulation Conference, Washington D.C., 1: 175-182.

ACKNOWLEDGMENTS

There were many people that helped me along the way to completing this work. I am indebted to my advisor Jan Cuny for her help and guidance throughout my graduate career. She gave me guidance when I needed it, let me explore when I wanted to, and I feel fortunate to have had the opportunity to work with her. I am also indebted to Roy Haggerty and Alphonce Guzha for their valuable collaboration throughout this research. I owe ineffable thanks to my parents for their unfaltering support throughout my academic career and to my girl Kelli for being there to help me through these challenging years. I send props to all my colleagues and friends that I've made along the way, and I thank Merlin for his relentless companionship throughout the preparation of this manuscript. The investigation was supported in part by grants from the National Science Foundation, ACI-0081487 and SBE-0318372.

To my family.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

Background and Motivation 1
Contributions of this Work 5
Organization of the Dissertation 6

II. RELATED WORK 7

Introduction 7
Approaches to Model Coupling 7
The Monolithic Approach 8
The Scheduled Approach 9
The Component Approach 10
The Communication Approach 12
Computational Steering 14
Summary 17

III. APPROACH OVERVIEW 20

Introduction 20
Creating a Coupled Model 20
Representing Model Interfaces for Coupling 24
Summary 28

IV. DESCRIBING COUPLING POTENTIAL 29

Introduction 29
What Is Coupling Potential? 29
The Basic Elements of Coupling Potential 30
Presentation of the Basic Elements 31
The PCICreate Software Assistant 36
How PCIs are Created 42
Creating the PCI 43
Instrumenting the Model Codes 48
Is There More to Coupling Potential? 54
The Coupled Model Study 54
Methodology 54
Results 56
Model Compatibility 58
Summary 61

Chapter Page

V. DESCRIBING COUPLED MODELS 62

Introduction 62
Overview 62
The Actions 64
The Send Action 64
The Update Action 67
The Store Action 69
Sending Data Between Models 71
Coordinating Data In Time 85
Implicit Temporal Coordination 85
Explicit Temporal Coordination 87
Examples 89
Example One 89
The Participating Variables 93
Creating the Coupling Description 94
Incorporating the Spatial Distribution of Physical Quantities 98
Incorporating Another Model 99
Example Two 103
The Participating Variables 104
Creating the Coupling Description 105
Simulating More Areas 108
Summary 111

VI. EXECUTING COUPLED MODELS 112

Introduction 112
Overview of the Runtime System 112
Compiling the Coupling Description 114
Send Actions 114
Store Actions 115
Update Actions 116
Example Compilation 116
Operation of the Runtime System 118
Starting a Coupled Model 118
How Couplers Store Values 118
How Couplers Execute Update Functions 119
Summary 121

Chapter Page

VII. CASE STUDIES 122

Introduction 122
Case Study: Simulating Stream-Aquifer Interaction 123
Coupled Model High-Level Design 124
Calculating Seepage 124
Implementation of the Reference Coupling 127
Implementation of the Interface Coupling 130
The Study Site 135
Evaluation of the Interface Coupling 136
Case Study: Watershed-wide Surfacewater Transport 139
Coupled Model Overview 141
The Coupled Model Inputs 142
The Coupling Description 149
Results 152
Case Study Summary 155
Case Study: Simulating Runoff-Aquifer Interaction 156
Motivation 157
Coupled Model Design 157
The Coupling Description 159
Results 166
Case Study Summary 168
Summary 169

VIII. CONCLUSIONS 170

APPENDIX

A. PCI COLLECTION 175
B. GUIDELINES FOR ANNOTATING MODEL CODES 186
C. ADDING CUSTOM DATA MAPPINGS 188
D. ADDING CUSTOM UPDATE FUNCTIONS 189

BIBLIOGRAPHY 191

LIST OF FIGURES

Figure Page

1. The comprehensive simulation incorporates many different processes 2
2. The comprehensive simulation accounts for interactions between streams 3
3. Model codes are scavenged to create a new model code 8
4. Model inputs and outputs are connected 9
5. Components are connected to create models 11
6. Models are instrumented so they can exchange data at runtime 13
7. The server acts as an intermediary 15
8. Two different coupling surfaces 22
9. The scientist works only with the coupling interfaces 24
10. A single PCI for each model is created and reused 27
11. A PCI for the model ModFlow 34
12. How recursion is handled in the PCI 36
13. Screenshot of the PCICreate application 37
14. The solution loop is grouped into a single block labeled "Group" 38
15. How a coupling point can appear as an edge or a block 38
16. Information about blocks can be viewed in the inspector 40
17. The inspector window shows the accessible variables 41
18. The process of creating a PCI 42
19. An example of an annotation 43
20. How interval analysis collapses a subgraph 45
21. How our algorithm collapses a subgraph 46
22. How a different placement of annotations affects the reduced graph 47
23. Early vs. late instrumentation of model codes 49
24. Scripts describe coupling-specific behavior 50
25. Source code of the generated accessor subroutine for the annotation shown in Figure 19 51
26. Interactions between hydrological systems 56

Figure Page

27. Summary of similarities and differences between the models 59
28. The coupling environment of PCICouple 63
29. The explicit depiction of Send Actions 65
30. An intra-model Send Action 65
31. The sent variable X is now accessible at Coupling Point B 66
32. The properties of the Send Action as they appear in the inspector 67
33. Source code for the built-in assignReal function 68
34. The properties of the Update Action in PCICouple 68
35. The properties of the Store Action in PCICouple 69
36. A simple data mapping between two models 71
37. Sending a variable to two different models 72
38. Sending data from two different models 72
39. The data mapping indicates there are three instances of the Stream model 73
40. A data mapping that indicates there are two instances of the Lake model 74
41. A data mapping that indicates there are two instances of each model 75
42. A data mapping that indicates that two instances send to a third instance 76
43. A data mapping that indicates there are five instances of the Stream model 76
44. Each instance can be identified by a unique number 77
45. The y variable is sent to only instance 2 of the Lake model 78
46. Array-level data mappings allow for a finer grain mapping 79
47. Format of data mapping input files 80
48. The meaning of variables differs across models 81
49. Third-party sources of information can be used to relate variables 81
50. The elements of the two arrays represent different spatial areas 82
51. A GIS can be used to relate spatial data 83

Figure Page

52. The physical space represented by the elements of an array can change between study sites 84
53. Implicit coordination by matching start times and step lengths 85
54. Specifying frequencies to resolve differences in time step length 86
55. Typical structure of a discrete event simulation 87
56. Illustration of the physical system simulated by SWMM 90
57. Water flux occurs through the unsaturated zone 91
58. Relationships between the water table, root zone, and the surface 91
59. Illustration of the physical system simulated by ModFlow 92
60. Subcatchments are irregularly shaped polygons 93
61. A single subcatchment superimposed on a single grid cell 94
62. The coupling description as shown in PCICouple 95
63. The source code for the setHead update function 96
64. The details of the Send and Update Actions 97
65. The default 1-to-1 data mapping 97
66. The area beneath the subcatchment is discretized as four cells 98
67. The array-level data mapping 99
68. How the array elements are combined and sent 99
69. Illustration of the physical system simulated by UEB 100
70. The updated coupling description 101
71. The source code of the addMelt function 101
72. The details of the Send and Update Actions 102
73. Illustration of the physical system simulated by GLEAMS 103
74. Lumped models do not support heterogeneity 104
75. The spatial distribution of the variable 105
76. The coupling description as it appears in PCICouple 105
77. The data mapping used by the Send Action 106
78. The details of the Send and Update Actions 106
79. The final coupling description 107
80. Details of the Update Action applied to the nutin variable 108

Figure Page

81. A field with four parts 109
82. An alternative data mapping indicating four instances 109
83. How values from multiple instances are combined 109
84. A field with nine parts 110
85. A data mapping indicating nine instances 111
86. Overview of the runtime system 113
87. Script for instance 1 117
88. Script for instance 2 117
89. Instances send values to couplers only 119
90. Couplers store values for their assigned instances 119
91. Function inputs are assembled by the coupler 120
92. Couplers use updaters to apply update functions 121
93. Illustration of the physical system simulated by DAFlow 123
94. Each subreach is associated with a single grid cell 125
95. Physical quantities that influence seepage 126
96. How the model codes were integrated, arrows indicate function calls 128
97. The coupling description as it appears in PCICouple 131
98. The source code for the calcSeepage subroutine 132
99. The source code for the tributary functions 133
100. The source code of the adjVolume function 134
101. The schematic for example application 1 135
102. Simulated streamflow at node 14 for each coupling 137
103. Comparison of aquifer head at a well located in row 7, column 10 137
104. Mack Creek gaging station in the Andrews Experimental Forest 139
105. Illustration of the physical system simulated by STAMMT-L 141
106. The physical stream network and its abstraction 142
107. The input file used by STAMMT-L 143
108. The stream network and forest boundary in ArcMap 145
109. The extended attribute table for the stream network in ArcMap 147

Figure Page

110. Script used to generate the value list 148
111. The generated parameter list (showing values for only 17 instances) 149
112. The coupling description 150
113. The ArcMap script used to generate the data mapping 151
114. The data mapping used in this case study (for only 17 streams) 152
115. Breakthrough curves of different streams 153
116. Comparison of the breakthrough curve of instance 12 with that of 11 and 17 154
117. Tenmile creek 156
118. Three subcatchments in the Tenmile Creek Watershed 158
119. Subcatchments superimposed on a grid 158
120. The coupling description 159
121. Source code for the setHead function 162
122. The data mapping relates the regular grid to the irregular subcatchments 163
123. Spatial units are numbered by array element or instance id 163
124. The script that generates the data mapping 164
125. The runtime environment provided by PCICouple 165
126. The precipitation input used by TopModel in all three cases 166
127. Comparison of the overland flow simulated by TopModel in each case 167
128. Comparison of the recharge simulated by TopModel in each case 168
129. A PCI for BioMOC 176
130. A PCI for Branch 177
131. A PCI for FourPt 177
132. A PCI for DAFlow 178
133. A PCI for GLEAMS 179
134. A PCI for SHAW 179
135. A PCI for ModFlow 180
136. A PCI for OTIS 181

Figure Page

137. A PCI for STAMMT-L 181
138. A PCI for SWAT 182
139. A PCI for SWMM 183
140. A PCI for TopModel 184
141. A PCI for UEB 185
142. A PCI for WASP 185
143. Available data mappings (left) and the mapping details (right) 188
144. The communication wrapper function for the setHead function 189
145. The update function list (left) and function details (right) 190

LIST OF TABLES

Table Page

1. A summary of frameworks that can be used for coupling 18
2. The basic elements of coupling potential 31
3. Reduction ratios for various hydrological models 47
4. Models used in the coupling study 55
5. Coupling-relevant variable characteristics identified in the study 58
6. Time and bandwidth measurements of each coupled model 138
7. The input parameters for STAMMT-L 144
8. Representative properties of streams of different sizes 144

CHAPTER I

INTRODUCTION

Background and Motivation

Modeling and simulation have become an essential tool in scientific investigation, complementing both the theoretical and experimental activities of science. The ability to simulate a dynamic system that is too difficult or costly, or even impossible to experiment with has been one of the most important technological advances for science. As a result, a great wealth of models exist today, informing nearly every scientific discipline, including biology, physics, geology, hydrology, and climatology. These models (also called computational models or computer models) vary greatly in their complexity and purpose, from spreadsheet-based financial prediction models that run on handheld computers, to earth-scale climate prediction models that run on the biggest and fastest computers in existence.

For the purposes of this work, we define a *model* to be the mathematical representation of some dynamic system along with its implementation as a computer program and all the associated input data and documentation. The term *model code* is used to refer specifically to the program source code of a model.

As our ability to accurately model individual physical phenomena has increased, the challenge now is to create simulations of complex, interacting

physical systems. In the hydrological community, for example, models of surfacewater, groundwater, rainfall-runoff, and transport are now being combined into simulations of the complete water cycle. Figure 1, for example, illustrates the various processes of the hydrological cycle that could be incorporated into a surfacewater-flow model.

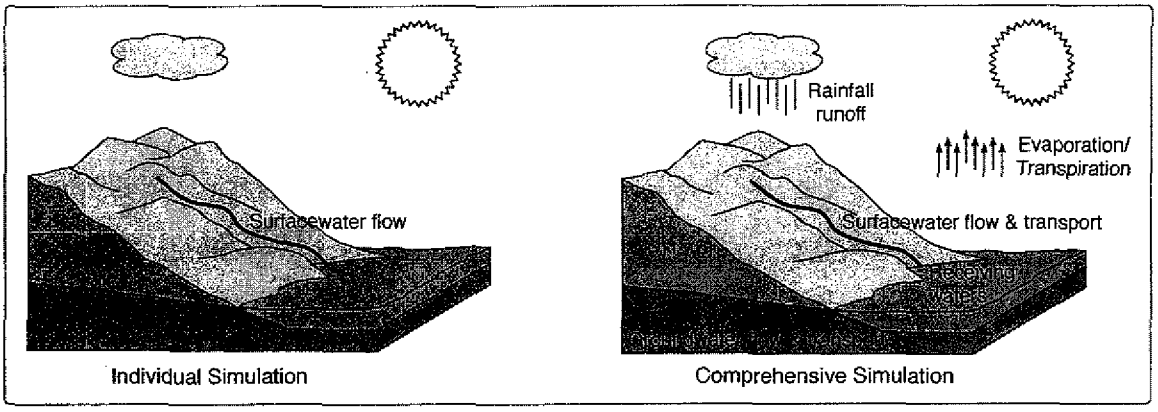

Figure 1. The comprehensive simulation incorporates many different processes.

In some cases, this combining of models can happen between instances of the same model linked together to cover interactions over larger geographic areas. Models that simulate limited areas could simply be applied repeatedly in order to simulate a wider area, but independent simulations preclude the ability to study the interactions between instances. For example, a model capable of simulating a single stream could be applied once for each stream of a network, but the interactions between the streams would not be accounted for in the simulations. A more holistic, accurate approach would be to combine the simulations into a single, larger simulation as illustrated in Figure 2.

This raises the question of how to achieve these more comprehensive models? One way is by developing entirely new models that encompass multiple phenomena across wide areas, but the time and expense involved in developing such models is prohibitive. Designing and implementing a model requires a combination of software engineering skill and domain expertise, and involves an ex-

tensive amount of verification and validation, often requiring field studies, throughout the entire development process.

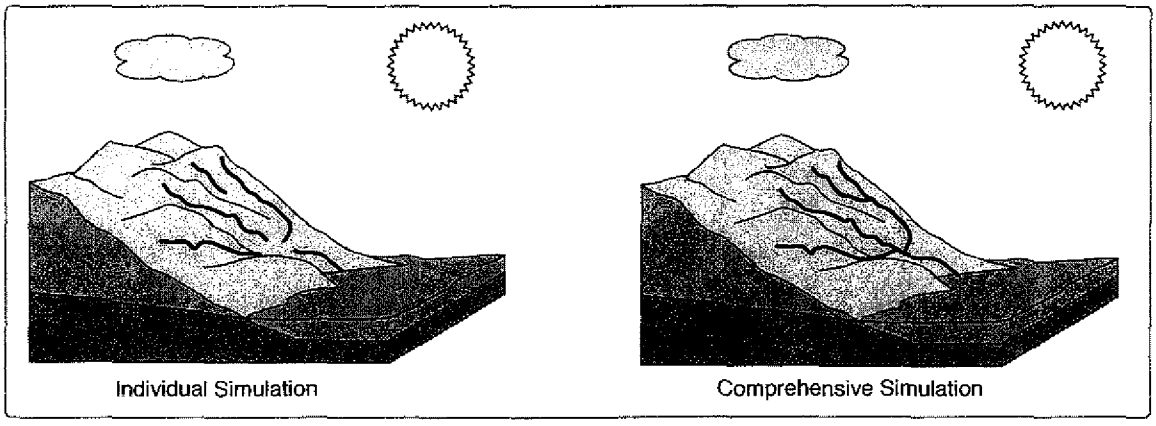

Figure 2. The comprehensive simulation accounts for interactions between streams.

The more complex the model, the more stifling these aspects become. For this reason, most computational models were developed as complex, special-purpose monolithic programs (Page et al. 1998) that simulate an isolated physical phenomena.

A better approach to creating new, more comprehensive models, is to reuse these existing models, combining them in a process called *model coupling.* The term *coupled model* is widely used although there is no common agreed upon definition. For the purposes of this work, we start with a general definition: a coupled model is a group of models executing together that have the ability to affect each other's computations. The reuse of software is a key principle of software engineering and is usually achieved by developing a set of simple components, or modules, that can be combined in different ways to create more complex components. Ideally scientists would couple their models by integrating the existing model codes, treating them as modular pieces that can be easily and quickly plugged together. This, however, can be a very difficult task when the legacy model codes themselves may be poorly understood, were not originally

designed to be coupled, and may be written in different programming languages for different computing platforms!

Despite the potential benefit of building new models from existing ones, model coupling is not a common practice in the scientific community because of the difficulties inherent in working with model codes. Reusing source code in general is difficult for many reasons. Not only are programs difficult to comprehend (a necessary part of any software reuse) (Rajlich and Wilde 2002) but the task of identifying useful source code fragments and integrating these source code artifacts that were not designed for reuse is challenging (Krueger 1992). This is especially true for model codes written in unstructured languages and languages that make extensive use of global data (e.g. Fortran). Reusing model code is particularly difficult because models are a unique class of computer programs whose design and use is intertwined with a great deal of domain-level theory outside the model code itself (Robinson et al. 2004). The task of understanding a model code requires relating the variables and calculations in the model code to the domain-level concepts with which they are associated. This makes the task of understanding how a model can be coupled to another, that is, understanding the *coupling potential* of a model, difficult.

The coupling potential of a model can be thought of as an abstraction that describes the characteristics of a model that in some capacity dictate how it can be coupled to another model (this is discussed in detail in Chapter 4). Model code is a poor representation of coupling potential because it includes every detail about how a model is implemented, most of which is not relevant to coupling, and the model code itself does not convey to the scientist its domain-level meaning.

Our hypothesis is that a better representation of coupling potential would enable the design of infrastructure that could more effectively support model

coupling. Such infrastructure could simplify the task of model coupling to the extent that it would be possible for scientists to quickly prototype coupled models. Our work seeks to design a more powerful representation of coupling potential and to test that representation by developing a model coupling environment for hydrological models based on it.

Contributions of this Work

The primary research contribution of this work is the development of a novel approach to model coupling that supports the fast-prototyping of coupled models. We call our technique the InCouple approach. To our knowledge this is the first work to focus on the representation of the model coupling interface as the basis of the approach. The model coupling interface is a high-level representation of a model that is used in the design of coupled models. We make three specific contributions. The first is

(1) *the design of a novel representation for model coupling interfaces: the Potential Coupling Interface (PCI).*

This representation serves four roles: it is a new form of metadata describing the coupling potential of a model; it is the vehicle for describing couplings; it is the basis for automatic source code generation; and it is a reusable representation.

The second contribution is

(2) *the design of a coupling language based on PCIs.*

This language is used to describe the behavior of coupled models, and includes support for resolving incompatibilities (spatial, temporal, etc.) between models. It may be applicable to other coupling frameworks.

The third contribution of this work is

(3) *the implementation and evaluation of a test environment for hydrology.*

The implementation was created as a proof-of-concept to demonstrate how the InCouple approach could be implemented. It was important to build the prototype to show that the design is usable in practice. Although our approach is general and applicable to nearly any scientific domain and nearly any kind of model, we chose hydrology as our prototype domain because the basic concepts of hydrology are generally accessible, and there is a wide variety of open source models available from many universities and government agencies. In addition, existing hydrological models typically simulate a particular subsystem of the hydrological cycle, making them obvious candidates for coupling. Most models in this field are serial (non-parallel), continuous simulations, some are spatially distributed, and others are *lumped*, meaning they do not account for the spatial variation of physical quantities (also called *lumped-parameter*).

Organization of the Dissertation

Chapter 2 reports on existing approaches to model coupling. Chapter 3 presents an overview of the coupling process and introduces coupling interfaces. Chapters 4 and 5 present a specification mechanism for coupling interfaces and the coupling language based on that interface. Chapter 6 explains how the runtime system executes coupled models based on coupling descriptions. Chapter 7 presents three case studies that demonstrate our approach, and Chapter 8 contains our conclusions.

CHAPTER II

RELATED WORK

Introduction

Comprehensive models can be created in a variety of ways. This chapter surveys existing approaches and emphasizes frameworks that were designed for coupling models, as well as those that can be used for coupling even though not specifically designed for it. In the next section, we describe each approach and classify them based on how the model codes are integrated. In the following section we discuss a related field, computational steering. We conclude with a summary that provides the context for our work.

Approaches to Model Coupling

The most basic way to integrate multiple model codes, called the *monolithic* approach, is to merge them into a single program. Some coupling-like techniques allow for limited interaction between models, and are collectively referred to as the *scheduled* approach. More sophisticated approaches involve frameworks designed to support model coupling. We use the term *frameworks* to refer to software systems that assist in designing software by providing a foundation upon which more complex and customized software can be built. With respect to model coupling, frameworks provide the building blocks to create coupled models. Some frameworks focus on enabling models to communicate, called *com-*

munication frameworks, while others focus on creating models from software components, called *component* frameworks. Here we discuss each of these four categories of approaches: *monolithic*, *scheduled*, *communication*, and *component*.

The Monolithic Approach

The monolithic approach to model coupling was first used to couple models (Swain and Wexler 1996; Jobson and Harbaugh 1999; Guo and Langevin 2002). In this manual process, the scientist creates a single program from the individual model codes, splicing together source code fragments with additional custom source code as illustrated in Figure 3.

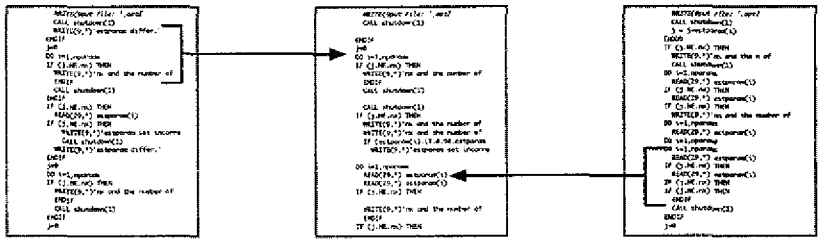

Figure 3. Model codes are scavenged to create a new model code.

Early monolithic hydrology couplings by the U.S. Geological Survey (Jobson and Harbaugh 1999; Swain and Wexler 1996), for example, merged the groundwater-flow model ModFlow (McDonald and Harbaugh 1988) first with the surfacewater-flow model DAFlow (Jobson 1989), and then with the more complex surfacewater-flow model Branch (Schaffranek, Baltzer, and Goldberg 1981). In both cases, the surfacewater-flow model codes were separated into subroutines which were then combined with the groundwater-flow model code to create a single model code. This monolithic approach had the advantage of being familiar to scientists, as it was not much different from the normal model writing process, that is, composing subroutines and writing source code. This approach also gave the scientist full control over all the coding details (control structure, memory allocation, data types, input/output file formats, etc.).

Although these couplings were very successful, the approach had significant drawbacks. The scientists performing the couplings needed a complete and detailed understanding of the constituent models and their model codes, which is often difficult to obtain: Legacy model codes are frequently complex, uncommented, and poorly documented. In addition, this whole process must be repeated from scratch by anyone wanting to replace one of the constituent models. The single combined model code is also difficult to work with from a software engineering point of view (testing, debugging, verifying, updating, etc.) since it is much larger than its constituent model codes, and improvements made to the original model codes must be repeatedly made to each coupled model code as well. For these reasons, scientists turned to more reusable techniques for coupling models.

The Scheduled Approach

In the scheduled approach, the models remain independent programs and do not affect each other as they are executing. A model is given a dataset and executed to completion to generate an output dataset. That output dataset is then given to another model (perhaps after some transformation) which uses it as input and is also executed through to completion. This process can then be continued with other models, and they can be executed concurrently if there are no dependencies between their datasets. This is illustrated in Figure 4.

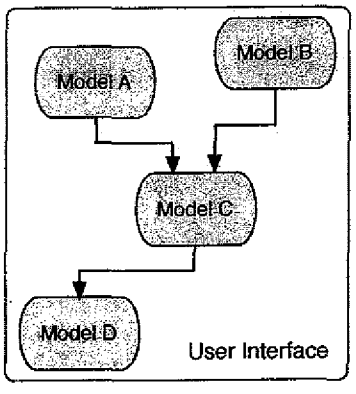

Figure 4. Model inputs and outputs are connected.

Scheduling frameworks support this process by providing an automated way for the user to select models and datasets, and then specify the distribution of datasets and the order of model execution (Simon; Akarsu et al. 1998; Whelan et al. 1997). One such framework, Le Select (Simon), uses a database-oriented approach in which both models and datasets are stored in geographically distributed databases and the user specifies the execution and data distribution through textual queries in a standard database query language. Others provide a visual interface to specify the order of execution of the models (Whelan et al. 1997; Akarsu et al. 1998). Models typically have different user interfaces and different input/output data formats making them difficult to use, especially for non-specialists. Some frameworks provide a standard user interface to each model (Neteler and Mitasova 2004). This requires that the original user interface source code be removed from each model and replaced with the common user interface source code. To address the issue of nonuniform data input and output formats, some frameworks require the user to perform this data transformation manually between model runs (Akarsu et al. 1998), while others require the user to change the model codes so that they use a standard data format (Whelan et al. 1997). This requires that all the input and output source code be removed and replaced with source code to access the common database and use its data types.

The Component Approach

The component approach to model coupling is similar to the monolithic approach in that the result of the coupling is a single model code, but differs in that rather than decompose the constituent model codes into blocks of source code designed for integration into another specific model code, the scientist decomposes the model codes into software *components*. Components are subroutines that are modular and reusable. We use the term *modular* to mean that the components can be written with little knowledge of the other components, and that they can be replaced independently without significant changes to the rest of

the program (Parnas 1972). We use the term *reusable* to mean that a component can be used in a variety of different situations without any changes made to it. Components possess a standard interface for invoking and passing parameters. The components are then recomposed by connecting their interfaces, as shown in Figure 5.

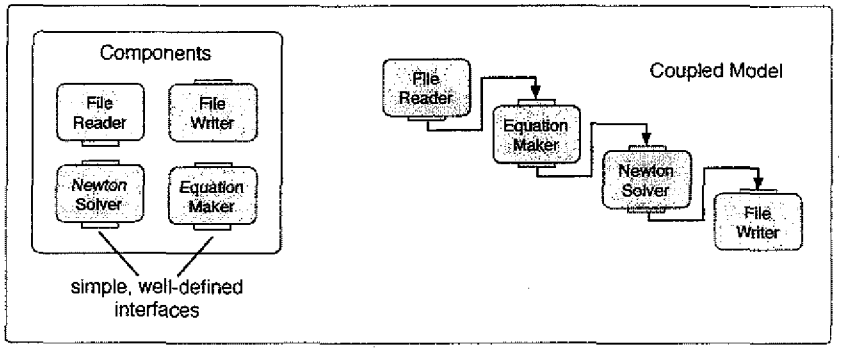

Figure 5. Components are connected to create models.

This can be thought of as a finer-grained scheduled approach, at the level of software components rather than models. The interfaces are generally simple and consist of a set of input data that must be supplied before the component can be executed, and a set of output data that is available upon completion. The computation that a component performs is encapsulated and hidden within the component. These components can be classified (Breunese et al. 1998) and organized in searchable libraries and the strict interfaces allow for automatic compatibility checking (Fox, Brogan, and Reynolds 2004). The specification of the component connections can either be through a visual environment (Piacentini 2002; Johnson et al. 2002; Gijsbers 2003; Blind et al. 2000; Leavesley et al. 1996; Ahuja, David, and Ascough 2004) or textual configuration files (Ford et al. 2004; Balaji 2002; Hill et al. 2004; Knox et al. 1997). Components are connected by specifying which component outputs map to which component inputs. This configurable dependence is a key feature of components. The component-based strategy is advantageous because simpler components are easier to test, debug, update, compare, and verify, and once the components are created they can be

easily assembled, reassembled in different ways, and can be reused in later compositions. In addition, frameworks can supply pre-made general purpose components for common operations (Piacentini 2002; Hill et al. 2004; Johnson et al. 2002; Gijsbers 2003; Blind et al. 2000; Leavesley et al. 1996). Projects that use the component approach differ in their interfaces and how the components can be connected. Some frameworks standardize the argument data types of the components (Hill et al. 2004; Johnson et al. 2002; Gijsbers 2003; Blind et al. 2000) while others allow no arguments and require *put/get* calls (from a custom library) within the component for data input/output (Ford et al. 2004; Piacentini 2002; Johnson et al. 2002; Leavesley et al. 1996), and some use a combination (Balaji 2002). Frameworks can be general purpose (Ford et al. 2004; Johnson et al. 2002) or apply to specific domains such as climatology (Piacentini 2002; Balaji 2002; Hill et al. 2004) or hydrology (Gijsbers 2003; Blind et al. 2000; Leavesley et al. 1996). Those that apply to specific domains typically support the transfer and transformation of domain-specific data types (grids, flux, etc.). Although the component approach addresses the issues of coupled model code complexity (by breaking the computations down into simple components) and reuse (by allowing easy reuse via standard interfaces), it still requires the scientist to have a complete and detailed understanding of the underlying model codes in order to rewrite them into components. Although coupled models would be easier to create from components than from scratch, such an approach to model coupling requires substantial reprogramming. For this reason, scientists turned to approaches that do not require such a substantial model code rewriting effort.

The Communication Approach

Using the communication approach, the underlying model codes remain independently executing programs that interact only by exchanging data via message passing during execution. Frameworks that use this approach can be classified by whether or not they include an independent application (a coupler)

that mediates the execution and communication between the models. The primary role of the coupler is to transform exchanged data, which typically involves data type conversions and mesh regridding (Joppich, Kurschner, and the MpCCI team 2005), but they also sometimes control the startup of the models or track the global state of the coupling as well (Sottile 2001). Frameworks that do not include a coupler are essentially libraries of data transfer and transformation routines customized for the data types (grids, flux, etc.) and communication styles (high-bandwidth, parallel, etc.) needed by models (Sklower et al.; Larson, Jacob, and Ong. 2005). Frameworks that do include a coupler have communication libraries that support direct model-to-model communication as well as model-to-coupler communication (Valcke et al. 2004; Blackmon et al. 2001; Beckman et al. 1998; Bettencourt 2002; Valcke, Guilyardi, and Larsson 2005; Joppich, Kurschner, and the MpCCI team 2005; Sydelko et al. 1999; Dahmann, Fujimoto, and Weatherly 1998). In either case, these libraries often require the scientist to convert the model data into a standard data type, which is then communicated. All of these frameworks require the scientist to instrument the model codes with library calls in order to send or receive data, as shown in Figure 6.

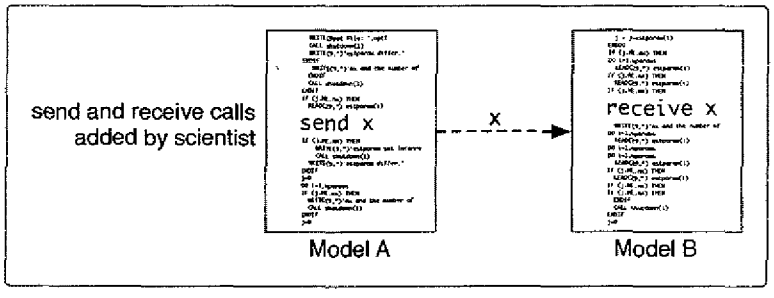

Figure 6. Models are instrumented so they can exchange data at runtime.

The user then writes configuration files that specify which models are to execute, and the data that is to be sent and received. The communication approach avoids the substantial model code rewriting required by the monolithic and component approaches, but the user still needs a complete and detailed understanding of the model codes in order to properly instrument them. Furthermore, the in-

strumentation is often specific to a single coupling, so the instrumentation process has to be repeated for each coupling. These frameworks also suffer from the problem that the instrumentation of the model code (e.g. adding a send() call which sends a value to another model) is separate from the specification of the coupling which is done in configuration files (e.g. specifying that one model will be sending a value to the coupler and another will be receiving a value from the coupler). This can make setting up a coupling error-prone because there is no way to perform consistency checks to ensure that the model codes are instrumented in a way consistent with the coupler configuration files. This typically isn't a serious problem when the projects that use these coupling frameworks are large (often climate-related) and specialists familiar with the models are available to instrument them, but this does become a problem for scientists in other domains seeking to create prototype couplings quickly and easily.

A related area of research called computational steering is presented next. This research is particularly relevant to the communication approach because both involve the instrumentation of source code to affect a model's behavior, and both utilize an intermediary application to facilitate communication.

Computational Steering

The ability to interact with potentially long-running scientific models during their simulations both facilitates performance enhancements that are difficult to automate and provides insight into the application's behavior. This is called *computational steering* (or interactive program steering, application steering, interactive steering). It enables scientists to observe and interact with a simulation during its execution, steering it as necessary. Common steering operations include modifying model parameters, adjusting load-balancing, changing algorithms used in the application, and starting/stopping/replaying some part of a simulation. Model coupling can be thought of as an instance of steering, because the con-

stituent models "steer" each other. Conversely, computational steering can be thought of as coupling a model to a scientist. In both cases, the inner state of a model is changing in response to events outside the model.

Steering is accomplished in two primary ways, either by swapping components in and out of an application as it is executing (such as changing the algorithm used in part of the simulation) or by manipulating an application's data (such as changing the value of a variable). In both cases, the state of the application is conveyed to the scientist, and the scientist can steer the application accordingly. These two functionalities, monitoring and user interaction, are typically supported in steering framework architectures through the use of an intermediary server (Jablonowski et al. 1993; van Wijk and van Liere 1997; Muralidhar and Parashar 2003; Gu et al.1998). The server collects monitoring data from the model and forwards it to the scientist, and forwards steering requests from the scientist to the application, as shown in Figure 7.

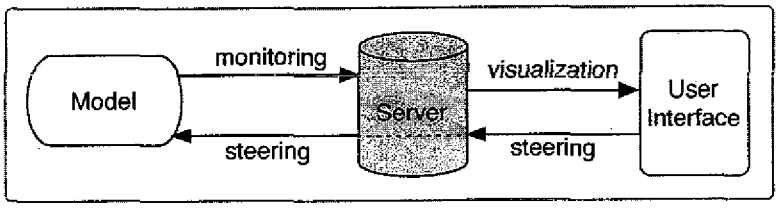

Figure 7. The server acts as an intermediary.

The monitoring data is presented in a variety of ways via the user interface and can range from simple variable-value inspection capabilities, to complex graphical visualizations provided by third-party software. This basic architecture is very similar to the communication frameworks presented earlier, where the server is performing a similar role as the coupler. Direct model-scientist interaction is avoided because a model may produce a great deal of monitoring data: sending this data to a local server allows the server to reduce the data via analysis before sending it on to the scientist, thus conserving bandwidth. This becomes more important in the case of distributed models being monitored by multiple, collaborat-

ing clients. In this case, a set of servers (usually one per machine) can be used to help distribute the data collection and analysis work which is then communicated through a central server to the scientist. Typically this analysis includes storing the data in a database (Rathmayer and Lenke 1997), reducing and filtering the data based on the needs of the user (Vetter and Schwan 1995), global event ordering (Gu et al. 1998), or distributed consistency analysis (Miller et al. 2001).

Program monitoring is the basis for steering: the user must know what the model is doing in order to intervene. Program monitoring is also important in debugging and performance analysis, especially in parallel and distributed application environments. In these cases, the overall behavior of the system is of interest, while computational steering focuses on just a few user-specified aspects of a model. The VASE system (Jablonowski et al. 1993) uses a graph to highlight the computational steering-related aspects, while the MOSS system (Eisenhauer and Schwan 1998) and DIOS system (Muralidhar and Parashar 2003) group application data into objects which are then presented to the user. In all the frameworks, this requires the user to directly instrument the model source code with function calls to a custom API library to register the variables with some form of server process. In most frameworks, the user defines an object that contains references to some set of variables, and then registers the object with the server (Rathmayer and Lenke 1997; Kohl and Papadopoulos 1998; Vetter and Schwan 1997). It is also possible to require variables to be registered individually (van Wijk and van Liere 1997) or to have the framework automatically determine the variables in scope at a particular point the program and present them to the user later on for selection (Jablonowski et al. 1993). In all cases, the user must decides where in the source code data is meaningful and readable and/or writable. Some authors claim that this is a straightforward task because these are easily identifiably places, such as before a main loop, or at the end of a main loop (van

Wijk and van Liere 1997), while others note that someone familiar with the source code must perform the instrumentation (Jablonowski et al. 1993). This instrumentation process is very similar to the instrumentation required by both the communication approach to model coupling and our approach.

When these points in the program are reached, the values of the registered variables are sent to another process, and hence, provide synchronous interaction between the application and server. In many applications of steering, it is very important to understand and minimize the effect that the steering system has on the application (i.e. minimize the perturbation of the application caused by steering). This can also be important in coupling frameworks if the models interact very frequently, incurring a great deal of communication time and dramatically slowing down the execution of a model.

Summary

This chapter surveyed a number of different frameworks that can be used for model coupling, organized into four different approaches. Clearly the component approach is an ideal way to construct new models if components are available, but the approach is impractical for coupling existing models. The communication approach though, allows existing models to be coupled with minimal changes to the model source codes. Since we are interested in model reuse, we focus on the communication approach in this work, and refine our use of the term *model coupling* to refer specifically to this approach. A summary of the frameworks discussed in this section is shown in Table 1.

Existing communication-style frameworks support the transfer and transformation of data between models well, assuming that the scientist knows which variables should be exchanged between the models, and what those variables mean. For example, if an air temperature variable should be sent from an atmosphere model to a sea-ice model, it is well known how to efficiently send and re-

ceive the data, and how to transform it in a variety of ways, from simple unit and data type conversions, to complex spatial regridding (Joppich, Kurschner, and the MpCCI team 2005).

Table 1. A summary of frameworks that can be used for coupling.

Project Name	Framework Type	Visual UI	Communication Type	Domain	Languages Supported	Reference
GCF	Component	No	Tuplespace	None	Fortran/C/C++	Ford et al. 2004
PALM	Component	Yes	Tuplespace	Climate	Fortran/C/C++	Piacentini and the PALM group 2002
FMS	Component	No	Tuplespace/Args	Climate	Fortran	Balaji 2002
ESMF	Component	No	Argument/Impl	Climate	Fortran	Hill et al. 2004
SciRun	Component	Yes	Implements	None	C/C++	Johnson et al. 2002
OpenMI	Component	Yes	n/a	Hydro	Fortran/C/C++	Gijsbers 2003
GF	Component	Yes	n/a	Hydro	C++	Blind et al. 2000
WISE	Component	No	n/a	Ecology	Fortran	Knox et al. 1997
CCA	Component	No	Arguments	None	Fortran/C/C++/Java	Armstrong et al. 1999
MMS	Component	Yes	Shared Memory	Hydro	Fortran/C	Leavesley et al. 1996
OMS	Component	Yes	n/a	None	Java	Ahuja, David, and Ascough 2004
DDB	Comm Lib	No	Custom Lib	Climate	Fortran	Sklower et al.
MCT	Comm Lib	No	MPI	None	Fortran	Larson, Jacob, and Ong 2005
GRISSLi	Comm Lib	No	MPI	None	Fortran/C	Ding, Munch, and Laux 1999
OASIS	Comm	No	MPI/PVM	Climate	Fortran/C/C++	Valcke et al. 2004
CCSM	Comm	No	MPI	Climate	Fortran	Blackmon et al. 2001
PAWS	Comm	No	Nexus	None	C++	Beckman et al. 1998
MCEL	Comm	No	CORBA	None	Fortran/C/C++	Bettencourt 2002
PRISM	Comm	Yes	Custom Lib	Climate	Fortran	Valcke, Guilyardi, and Larsson 2005
MpCCI	Comm	No	MPI	Climate	Fortran/C/C++	Joppich, Kurschner, and the MpCCI team 2005
DIAS	Comm	No	CORBA	Ecology	Fortran/C/C++	Sydelko et al. 1999
HLA	Comm	No	Any	None	Java/Ada/C++/IDL	Dahmann, Fujimoto, and Weatherly 1998
n/a	Comm	No	Any	None	Fortran/C/C++	Shengsheng et al. 2005
Le Select	Scheduling	No	n/a	None	n/a	Simon
WebFlow	Scheduling	Yes	CORBA/GLOBUS	None	Fortran/C/C++	Akarsu et al. 1998
FRAMES	Scheduling	Yes	Files	Risk Analy.	Fortran/C/C++	Whelan et al. 1997
GRASS	Scheduling	No	n/a	Hydro	Fortran/C	Ahuja, David, and Ong 2004

What is not known however, is how to assist the scientist in identifying the relevant variables, understanding what they mean, and crafting the interaction between the models. Currently, the only way to acquire this information is through a time-consuming and tedious manual analysis of the model code by each scientist who wishes to couple the model. Furthermore, existing frameworks offer little or no support for finding models that can be coupled, or identifying and understanding the kinds of incompatibilities that exist between them. Our work presents a solution to these problems based on a novel representation of model interfaces, customized for coupling, that captures this essential information in a reusable way. In this way, our approach focuses on the knowledge necessary to couple models. Our work does not seek to replace any of these existing frameworks, but rather, it complements these frameworks by assisting the scientist in additional tasks. Our representation could be used in conjunction with these frameworks to support the scientist in the task of designing a coupled model, after which the framework would be used to execute it.

Our goal is to enable scientists to easily experiment with a variety of couplings before investing in recoding efforts. Like communication frameworks, we support the use of annotated legacy model codes rather than recoding. Like component frameworks, we promote modularity and reuse, allowing the scientist to plug together models that have been made available within the framework. We support all aspects of model coupling (identifying appropriate models, specifying their interactions, integrating their model codes, and executing the coupled model), while allowing the scientist to work at a high level of abstraction without becoming enmeshed in low-level model code details. Before presenting the design of the interface, the coupling process that we intend to support is explained in detail, along with an introduction of how the use of model interfaces can simplify the process.

CHAPTER III

APPROACH OVERVIEW

Introduction

In this chapter we first present the process by which coupled models are created, and then discuss how the process can be supported through the use of a better representation for model interfaces for coupling.

Creating a Coupled Model

Before we can talk about support for model coupling, we must be clear about the process we are supporting. It is based on the communication approach introduced in Chapter 2. We describe the process in terms of six steps to which we refer throughout this text.

As in any modeling task, the process begins when the scientist identifies the physical system of interest, sometimes within a specific study site. The first step toward creating a coupled model is to obtain a set of models that collectively simulate these physical systems. In some cases the scientist may possess expertise with a specific model and want to couple another model to it, in other cases a group of specialists in one model may wish to collaborate with scientists in another domain working with a different model, or a scientist may need to collect all the models needed in the coupling.

Step 1. What models should participate in the coupling?

These models can be found by searching various agencies' online model repositories, such as the ones provided by the U.S. Geological Survey[1], and the U.S. Environmental Protection Agency's Center for Exposure Assessment Modeling[2]. These sites though, only give short descriptions of each model, requiring the scientist to download each one and review its documentation in order to determine whether or not a model is appropriate for his/her study. Some model repositories offer more detailed information (Plentinger and Penning de Vries 1996; Smith et al. 1997; Benz, Hoch, and Legovic 2001), but these still lack information about how the models can be coupled, complicating the task of finding appropriate models. Once the models have been collected, the scientist must understand what physical processes are simulated by each model, and how those processes affect each other.

Step 2. What interactions between the physical processes are to be studied and along what physical boundary do these interactions occur?

The scientist must identify how the physical processes simulated by each model interact with the processes simulated by the other models. The interactions - how the physical quantities of each process influence each other - can be unidirectional or bidirectional and they can take place continuously or only at discrete points in time. Often the physical quantities themselves are spatially distributed (in the physical study site), and influence each other across a common physical boundary. In this step, this boundary, which we call the *coupling surface,* is identified. Figure 8 shows an example of two different coupling surfaces. In the figure, A, B, and C represent modeled spaces, and 1 and 2 are coupling surfaces between them. At one extreme the coupling surface is an adjacency, labeled 2 in

---

[1] www.usgs.gov/software/

[2] www.epa.gov/ceampubl/

the figure; at the other, it is a complete overlap, the entire physical space simulated in both models, labeled 1 in the figure.

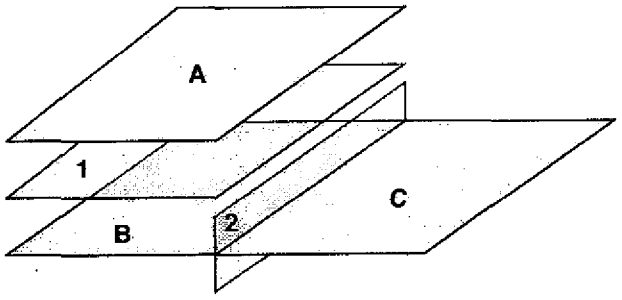

Figure 8. Two different coupling surfaces.

For the cases in between, the coupling surface is an overlapping region between the two modeled physical spaces.

Once the systems' interacting physical quantities and coupling surfaces have been identified, the next step is to look at the variables in the model codes that represent these quantities.

> Step 3. Which variables in each model represent the physical quantities involved in the interactions, and what are their syntax and semantics?

The variables that represent physical quantities (called *modeled quantities*) can be identified through inspection the model's documentation and/or its source code. We use the term *state variable* to refer specifically to a variable that represents some physical quantity. Model documentation typically includes lists of important state variables, describing what physical quantity they represent, their semantics (units and how they are spatially distributed), and their syntax (data type and shape). Often though, inspection of the source code is necessary to fully understand the syntax and semantics of a variable since available documentation is sparse.

Once the variables of interest are understood, the locations within each model code where they can be accessed by other models must be identified.

> Step 4. At what locations in the model codes should the state variables be exchanged between models?

Any point in a model code where a state variable is in scope is a point where that variable could be exchanged with another model. Typically though, the variable contains meaningful values at only a subset of these points. Choosing these critical points may be straightforward for the model author, or someone familiar with the model code, but for those not familiar, this task requires an exhaustive analysis of a model code. Further discussion of locating state variables in a model code is presented in Chapter 4.

Once the locations for accessing the relevant state variables are chosen, the next step is to specify precisely how the state variables affect each other.

> Step 5. What is the quantitative relationship between the variables of each model?

The values of state variables may need to be transformed before they can be used to affect each other. These transformations may be general purpose, or specific to a particular group of variables. General purpose transformations range from simple unit conversions to complex spatial regridding, and are often provided by coupling frameworks. Special purpose transformations are often domain-specific or are necessary to convert between custom data types, and must be provided by the scientist, usually in the form of additional source code.

The final step in creating a coupled model is to instrument the model codes to enable them to communicate.

Step 6. How should the model codes be instrumented to carry out the desired interaction?

Here, the scientist instruments each model code by adding new source code that allows the models to communicate. This is usually done in conjunction with a *coupling framework*. The scientist places calls within each model code to an API provided by the framework. Each model is then recompiled. Its execution is then typically handled through the framework and requires that the necessary configuration files for the framework also be prepared.

This section has stepped through the process of creating a coupled model using the communication approach. As evident from this discussion, existing coupling frameworks support only steps 5 and 6. The next section describes how the entire process can be supported through the use of a novel representation for model interfaces for coupling.

Representing Model Interfaces for Coupling

The motivating hypothesis behind this work is that model codes are a poor representation of coupling potential and that a better representation would enable infrastructure that could broaden the use of model coupling throughout the scientific community. We propose an interface-based approach where the scientist works exclusively in terms of a novel *coupling interface* to design and execute coupled models, freeing him/her from working directly with the model codes. This is illustrated in Figure 9.

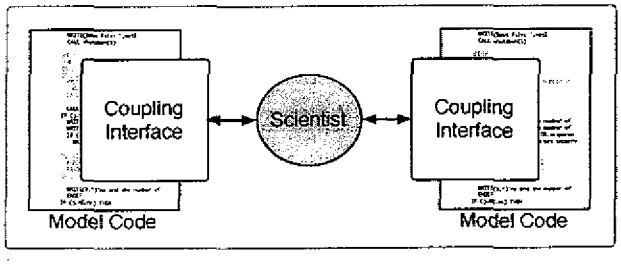

Figure 9. The scientist works only with the coupling interfaces.

The coupling interface is a high-level metadata description of a model that describes how a model can be coupled to any other model, that is, it describes a model's coupling potential and is created only once and then reused thereafter. We call our coupling interface the Potential Coupling Interface (PCI).

Traditional model documentation does not sufficiently describe the coupling potential of a model. It typically includes descriptions of the model and its limitations (variable description lists, simple flow graphs, domain diagrams, details about how space and time are abstracted, narratives, figures, equation descriptions, and references to related work) as well as information on executing the model (sample input and output, and hardware and software requirements) (Trescott, Pinder, and Larson 1980; Haggerty and Reeves 2003). Although this information is necessary, it is not sufficient for coupling. For coupling, the user must know which state variables can be accessed and where in the model codes those accesses can occur: Can a specific state variable be modified at any point in a simulation, or only during an initialization phase? Can a set of data be read by another model on every iteration of a solution loop, or is it only meaningful after the solution has converged? If two models use each other's state on each time step, how are their time steps to be coordinated? The Potential Coupling Interface succinctly captures this information and serves four primary roles. The PCI is:

1. a form of model metadata that conveys a model's coupling potential,
2. a vehicle for visually describing the behavior of coupled models,
3. the basis for automatic generation of model code instrumentation, and
4. a reusable interface allowing coupled models to be quickly prototyped.

Each of these roles facilitates the coupling process just described. The PCI augments existing model documentation with a complete description of the coupling

potential of a model. It can be added to a model's documentation and distributed with the model.

The PCI can augment emerging model metadata standards as well. In order to realistically share models on a large scale, scientists have begun to define metadata standards for model descriptions (Hill et al. 2001) allowing models to be organized, searched, and compared, similar to existing metadata standards for geographic data (Federal Geographic Data Committee 1998). CAMASE (Plentinger and Penning de Vries 1996), SOMNET (Smith et al. 1997), and ECOBAS (Benz, Hoch, and Legovic 2001), for example, provide online metadatabases with entries that describe a model's general characteristics, its scientific and technical specifications, contact information, and domain-specific information. By incorporating the PCI into these metadata standards and the repositories based on them, models can be organized, searched, and compared in terms of their coupling potential, simplifying, and possibly automating, the task of finding groups of models to couple (Step 1 of the coupling process).

Since the PCI exposes all the aspects of a model that are relevant to coupling, this representation is a sufficient basis for describing coupled models. This simplifies Steps 2, 3, and 4 of the coupling process because these steps require the scientist to expend a great deal of effort working with the model code and reviewing various forms of model documentation to locate the necessary coupling-related information. All of this information though, is clearly and concisely presented by the PCI. Just as coupling frameworks support the transformation of data between models (Step 5 of the coupling process), so too could a framework based on PCIs.

Since models are not written with the ability to communicate with each other, the model codes must be instrumented with additional source code that enables them to do so. We do not want the scientist to perform this task due to

the difficulty (and likelihood of error) involved in instrumenting model codes directly. Therefore, the model codes must be instrumented automatically. For this reason, the PCI is created directly from the model source code so that there is a direct correspondence between the interface and the underlying model code. This allows the interface both to generate source code automatically, since it is aware of the data types and variables of a model, and to instrument the model codes with the generated source code. Through the use of automatic instrumentation, Step 6 of the coupling process is fully automated.

A key advantage of an interface-based approach is that the interface for a model need only be created once, and after that, can be used in any coupling of that model. This process is shown in Figure 10.

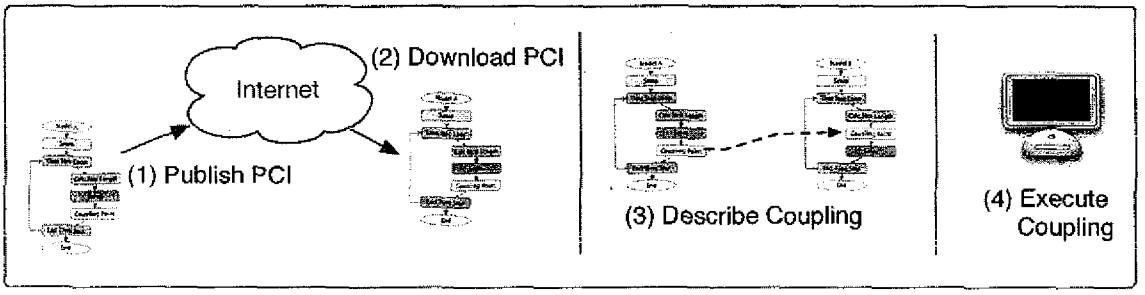

Figure 10. A single PCI for each model is created and reused.

Once the PCI has been created, it is published on the internet along with its associated instrumented model code, and possibly the coupling-ready model executable (1). When a scientist wishes to use the model in a coupling, s/he retrieves the PCI and the coupling-ready executable from the internet (2). The scientist then describes the coupling in terms of the model PCIs in a custom coupling environment (3) (further described in Chapter 5). Finally, the coupled model can be executed (4). Through the use of reusable interfaces, the time and effort required to construct coupled models is dramatically reduced.

Summary

This chapter explained the six steps in the process of creating a coupled model. The concept of using model interfaces for coupling was then presented, and the four primary roles of our novel coupling interface, called the Potential Coupling Interface, were introduced. The use of the PCI supports the entire process of creating a coupled model and does so in a reusable way. So far we've only said that the PCI describes the coupling potential of a model, but not how. The next chapter investigates the concept of coupling potential and identifies specifically what it entails.

CHAPTER IV

DESCRIBING COUPLING POTENTIAL

Introduction

This chapter presents the Potential Coupling Interface (PCI). The PCI *must describe the aspects of a model that affect and dictate how it can be coupled* to other models. What characteristics of a model must be exposed? How are those characteristics best described with minimal effort on the scientist's part? What is the best and most intuitive way to present them so that scientists can use them? This chapter gives our answers to these questions. The first section motivates and describes the design of the PCI and our process for creating them. The second section reports on the results of a coupled model study that we conducted in order to better understand coupling potential and how it differs across typical models in our domain.

What Is Coupling Potential?

We developed the PCI by first investigating a small number of case studies of existing and hypothetical coupled models. These studies identified the basic elements of coupling potential and are described in the next section, along with a software tool that is used to create PCIs.

The Basic Elements of Coupling Potential

From our initial case studies (Bulatewicz, Cuny, and Warman 2004), the purpose of model coupling was clear: to allow the state of a model to affect, and be affected by, the state of another model. The state of a model is the collective state of all its variables. So, model coupling allows the variables of a model to affect, and be affected by, the variables of another model. In the case studies though, only a small number of the variables of each model participated in the coupling: the ones that had meaning outside the model code itself. They were usually representative of physical quantities. All the other variables had no meaning outside the model, and were therefore irrelevant with respect to model coupling. Thus the first basic element of the coupling potential of a model is its state variables. These have to be specified by the PCI designer.

The values of these state variables are continually changing throughout the execution of a model code and they do not necessarily possess valid values at all points. For example, state variables typically possess valid values at the start of each time step, and at the end of each step where they are often saved as output. Within the time step loop or before the model inputs are read, though, state variables may possess partial or inconsistent values. Thus, it is not sufficient to simply know what the state variables of a model are: to use them within a coupling one must also know where they have meaningful values. This is the second basic element of coupling potential: the locations within a model code where state variables possess meaningful values. These also have to be specified by the PCI designer.

Different locations within a model code are reached different numbers of times throughout the simulation. The number of times that a location is reached during a simulation dictates how many times the state variables in scope at that location are accessible to other models for reading or writing. For example, program statements located within the initialization phase of a model code are

reached only once in each execution of a model, while statements located at the start of the time step loop are reached once on each iteration of the time step loop. In this way, control structures such as loops and conditionals dictate the accessibility of state variables. This is the third basic element of coupling potential: the control structure surrounding places where state variables possess meaningful values. These surrounding control structures are determined from the model source code, although the user may further simplify them.

The three basic elements of coupling potential are thus summarized in Table 2. Our initial design of the coupling interface describes these basic characteristics of a model code.

Table 2. The basic elements of coupling potential.

Basic Elements of Coupling Potential
State variables of a model
Locations in a model code where state variables possess meaningful values
The control structure surrounding these locations

Presentation of the Basic Elements

In order for an interface-based approach to model coupling to be practical, the coupling interface must intuitively and clearly describe all the elements relevant to coupling to the scientist. These elements are characteristics of a model code, and thus, the coupling interface must be derived from that source code. Since the coupling potential describes various aspects of the model code to the scientist (lists of variables, places where state variables have meaningful values and the control structure surrounding these places), it is an aid to *program comprehension*. The field of program comprehension seeks to understand how people comprehend programs, and how that knowledge can be used to develop ways to facilitate the process of understanding programs. Therefore, to guide our design of the presentation of the PCI, we employed research from the field of

program comprehension (also called software comprehension or program understanding) and present the relevant research throughout this chapter as appropriate. In this section we begin by showing an example of the presentation of the PCI, and then we discuss the design with respect to the basic elements described above and the field of program comprehension.

Our aim is to develop a representation that allows all coupling relevant aspects of a model to be readily understood by scientists who are not familiar with its source code, while at the same time maintaining a correspondence with the underlying program that is sufficient for automatic instrumentation and source code generation. In order to encourage the participation of the original programmers, we want our representation to be easy to create. To accomplish these goals, we base our representation on control flow graphs (CFGs) in which blocks (often called *nodes*) represent sections of source code and directed edges represent the flow of control between them. Note that we are using an abstraction of a CFG: in a standard CFG there is one block for each straight-line section of source code called a *basic block*, while in our use of CFGs, each block may represent many basic blocks.

We chose to use the CFG as the basis of the presentation of the PCI for several reasons. First, it is similar to the control flow graphs and flow charts that are commonly used in model documentation, so it is a representation of a model code that is already familiar to scientists. Second, it is a visual description, which the program comprehension literature suggests as a mechanism to facilitate comprehension (Storey et al. 1997). Since the PCI must assist the scientist in understanding a model code, the use of a visual representation for the PCI is important.

The complete control flow graph of a model often consists of thousands of blocks, most of which represent source code that performs very low-level tasks

that do not have any domain-level significance and hence are irrelevant to coupling. These low-level tasks include calendar calculations, equation solving, file input/output, etc. Although it is important for the scientist to know where some of these tasks are carried out, such as where input parameters are read, it is not necessary (or helpful) to describe every statement in the PCI. For this reason, we use a combination of automated and user-assisted mechanisms to reduce the CFG size (explained further in the next section). With these mechanisms, the complete graph is reduced to a smaller graph in which the uninteresting parts of a model code are grouped into blocks, hiding the irrelevant details and highlighting the locations of coupling points. We expect the reduced graphs to generally be small in size, and in the model codes we considered, the number of blocks ranged from approximately 10 to 30 (with 10 to 20 coupling points).

The VASE visualization system (Jablonowski et al. 1993) used CFGs in a similar manner to specify breakpoints for source code visualization. VASE graph reduction was not automatic but was specified by the user who demarcated blocks of source code that were to be coalesced; s/he then identified breakpoints by selecting CFG edges.

As an example, consider the PCI shown in Figure 11. It represents ModFlow (McDonald and Harbaugh 1988), a widely used groundwater-flow model developed by the U.S. Geological Society. ModFlow is written in Fortran and has approximately 10,000 lines of source code. We step through the process of creating this PCI in the following section. Appendix A presents additional PCIs for other hydrological models.

In our PCI, each dark arrow represents a *coupling point*, that is, a place in the model code where a set of state variables is accessible (coupling points can also appear as blocks in the PCI by "expanding" these arrows). Only the state variables that possess meaningful values at that point in the model code are ac-

cessible at a coupling point. The control structure (e.g. loops) around coupling points is clearly visible in the graph.

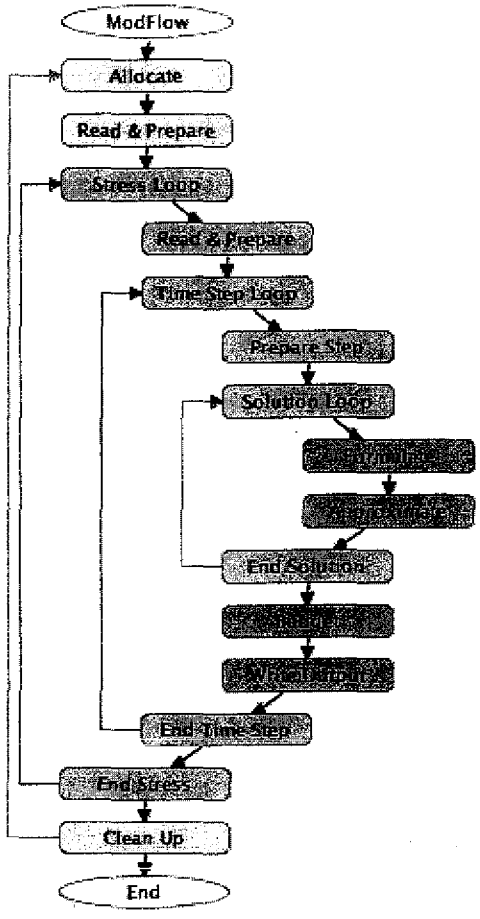

Figure 11. A PCI for the model ModFlow.

A key theme in the program comprehension literature is the different kinds of knowledge that are used and constructed when understanding programs. Of particular relevance to model interfaces for coupling is the difference between knowledge about the low-level details of a particular model code, and general knowledge about the problem domain and the field of modeling and simulation. Computational models are a unique class of programs because there is a great deal of information associated with them outside the model code itself, most of it at the domain-level. A central role of the PCI is to relate the domain-level knowl-

edge about a model to the constructs in the model code with which they're related.

Two common methods of facilitating knowledge acquisition in program comprehension have direct manifestations in the PCI: beacons and rules of discourse (Storey, Fracchia, and Müller 1997). Beacons are cues in a source code that index directly to knowledge, such as naming a routine *sort* to indicate its purpose. Similarly, the block colors and labels/descriptions act as beacons in the PCI - each block can be colored according to the role that its representative section of source code plays in the model, such as initialization, time step loop, or computation.

Rules of discourse are programming conventions, such as coding standards or mnemonic naming. With respect to model codes, the overall organization of the model code usually adheres to a typical structure depending upon the kind of model. In continuous simulations, for example, convention suggests that there is an initial setup and input phase, followed by a loop that solves a set of equations, and then concludes by outputting the results. A scientist's knowledge of how different kinds of models are typically structured can be directly and visually applied to a PCI, making the PCI immediately informative.

Other low-level aspects of a model code have little or no domain-level association, such as the functional decomposition of the statements of a model code. For this reason, the flow graph of the PCI is not structured according to how the subroutines of a model code are organized, but rather, the PCI presents a cohesive view of a model code, hiding details of the underlying functional decomposition. Subroutines that possess coupling points are inserted into the graph at each place where they are called unless they are recursive (i.e. it calls itself). If a subroutine is recursive, then it is not inserted at the recursive call. Similarly, if two subroutines are indirectly recursive (i.e. each one calls the other),

then the subroutines are not inserted at the (indirectly) recursive calls. For example, consider two subroutines, A and B, that call each other, and A is called from subroutine C, as shown in Figure 12 (top).

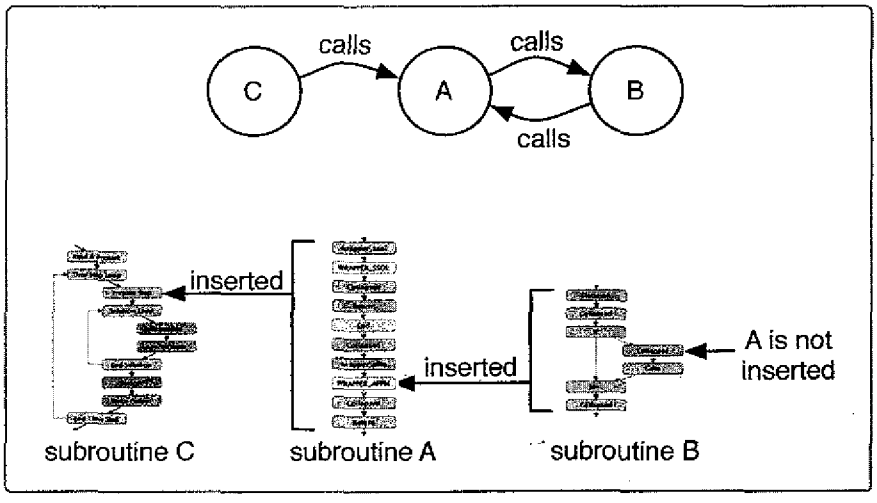

Figure 12. How recursion is handled in the PCI.

In the PCI, subroutine A is inserted where it is called in subroutine C, and subroutine B is inserted where it is called in subroutine A. But, subroutine A is not again inserted where it is called in subroutine B, as shown in Figure 12 (bottom).

The PCICreate Software Assistant

To assist the scientist in the creation of PCIs, we have developed an application called PCICreate that automatically converts an annotated model code into a simplified control flow graph and then allows the user to edit the graph and incorporate important high-level information into it. Model code annotation is explained in the next section. PCICreate is written in Java, making it usable on different computing platforms, and it uses the open source JGraph component (Benson 2006) to display the flow graphs. The implementation supports only Fortran source codes, but can be extended to support any programming language. A screenshot of the application is shown in Figure 13.

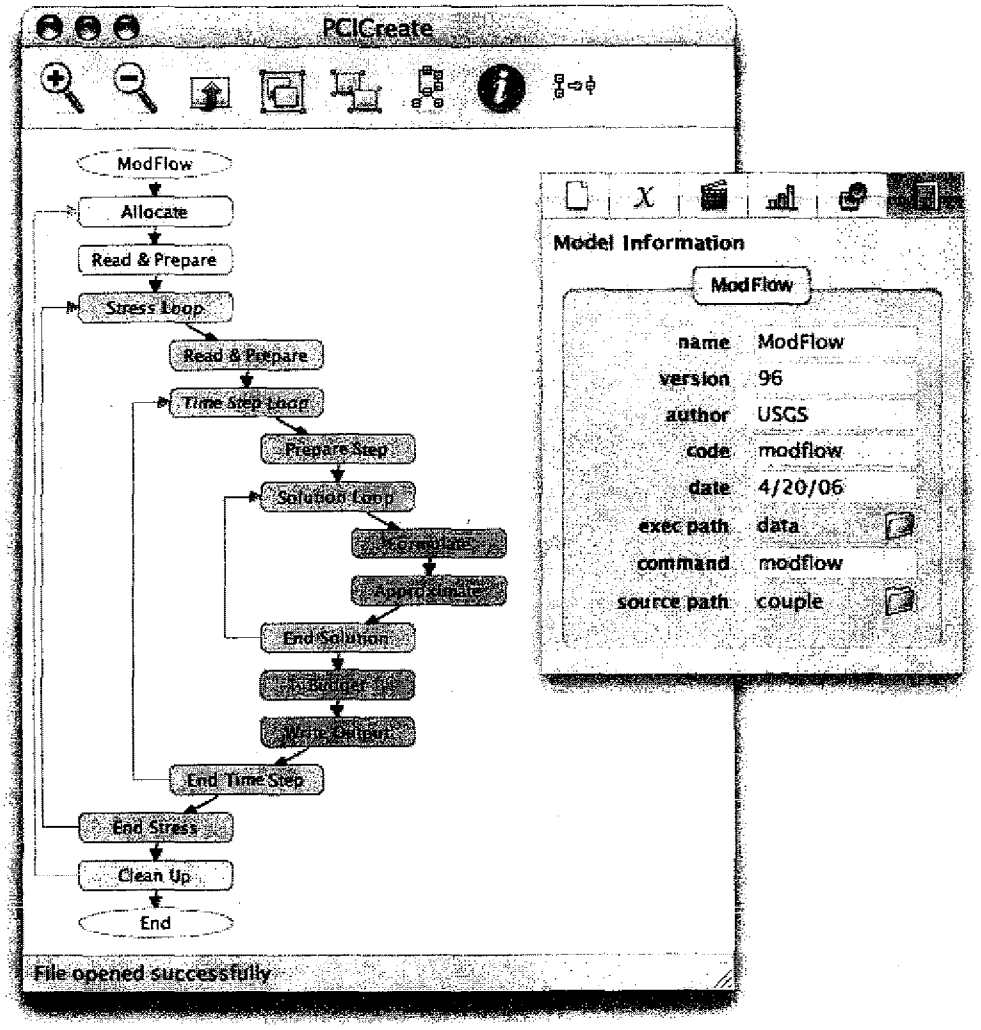

Figure 13. Screenshot of the PCICreate application.

The large editor window displays the PCI, and the smaller inspector window shows different kinds of information about the PCI and its parts. In the figure, the inspector is showing the model's general information, which has been entered by the PCI creator. The buttons along the top of the editor window allow the PCI to be viewed and manipulated in a variety of ways. From left to right, the buttons are: *zoom in, zoom out, show/hide coupling point, group blocks, ungroup blocks, auto-layout, show/hide inspector window,* and *collapse blocks.* Both group and collapse reduce the number of blocks in the graph, but the difference between them is that the collapse operation changes the structure of the PCI by merging

the set of blocks into a single block (which cannot be undone), while the group/ upgroup feature affects only the appearance of the PCI, but does not change the actual structure of the PCI. An example of grouping a set of blocks is shown in Figure 14.

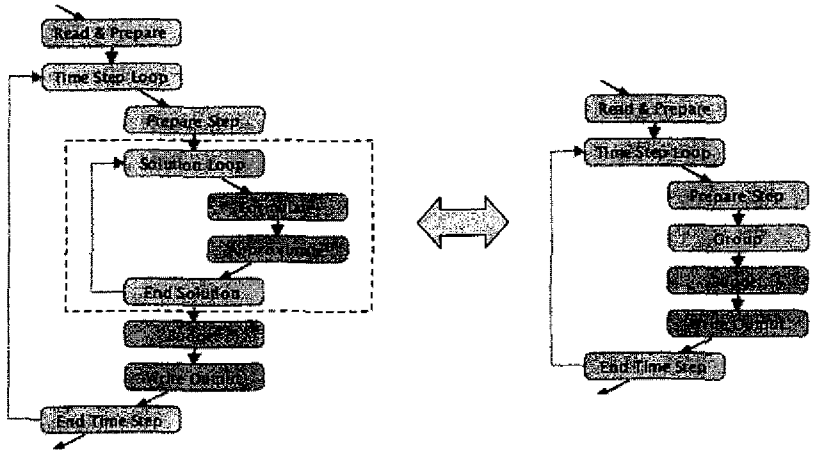

Figure 14. The solution loop is grouped into a single block labeled "Group".

The show/hide coupling point button allows the scientist to toggle the appearance of a coupling point between a dark arrow and a block (it is easier to see the coupling points of interest in a PCI by having them appear as blocks). An example is shown in Figure 15.

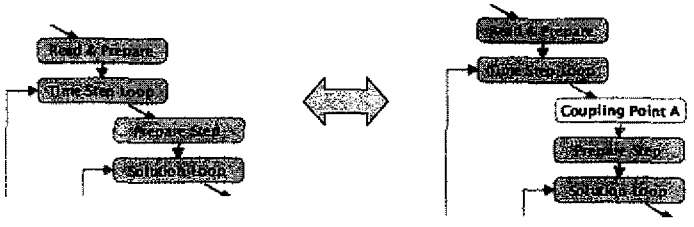

Figure 15. How a coupling point can appear as an edge or a block.

The placement of the blocks can be automatically arranged via the auto-layout button (the layout of the graph in Figure 13 was automatically performed), and the placement of individual blocks and edges can be changed by dragging the blocks. Menu items provide a way to save and load PCIs, generate the instrumentation, and save an image of a PCI.

Storey et al. have proposed guidelines for the design of program comprehension tools (Storey, Fracchia, and Müller. 1997). Those guidelines include:

- Indicate syntactic and semantic relations between software objects; both low-level source code as well as high-level relationships should be shown via graphs where nodes represent source code and arcs represent relations.

- Reduce the effect of delocalized plans (explanations of isolated, partial parts of a program); this can be avoided by not allowing any representation of the program to be isolated from other representations.

- Provide abstraction mechanisms; users should be able to create their own abstractions and label them to reflect their meaning and in many tools subgraphs can be collapsed into a single composite node.

The design of PCICreate adheres to each of these recommendations. The syntactic relationship between the blocks is shown in the graph structure, and the semantic relationship is indicated by the block colors and textual descriptions viewable in the inspector window. The label of each block quickly conveys the purpose of the section of model code that it represents. There is a single representation of the model code, the flow graph, so delocalized plans are avoided. Blocks in the graph can be grouped together and later ungrouped, providing a way for the user to abstract the graph as s/he desires. Information is associated with blocks and coupling points, and can be viewed via the inspector window. In Figure 16, the coupling point immediately following the "Time Step Loop" block has been made visible as a block and selected to show the general information about it in the inspector.

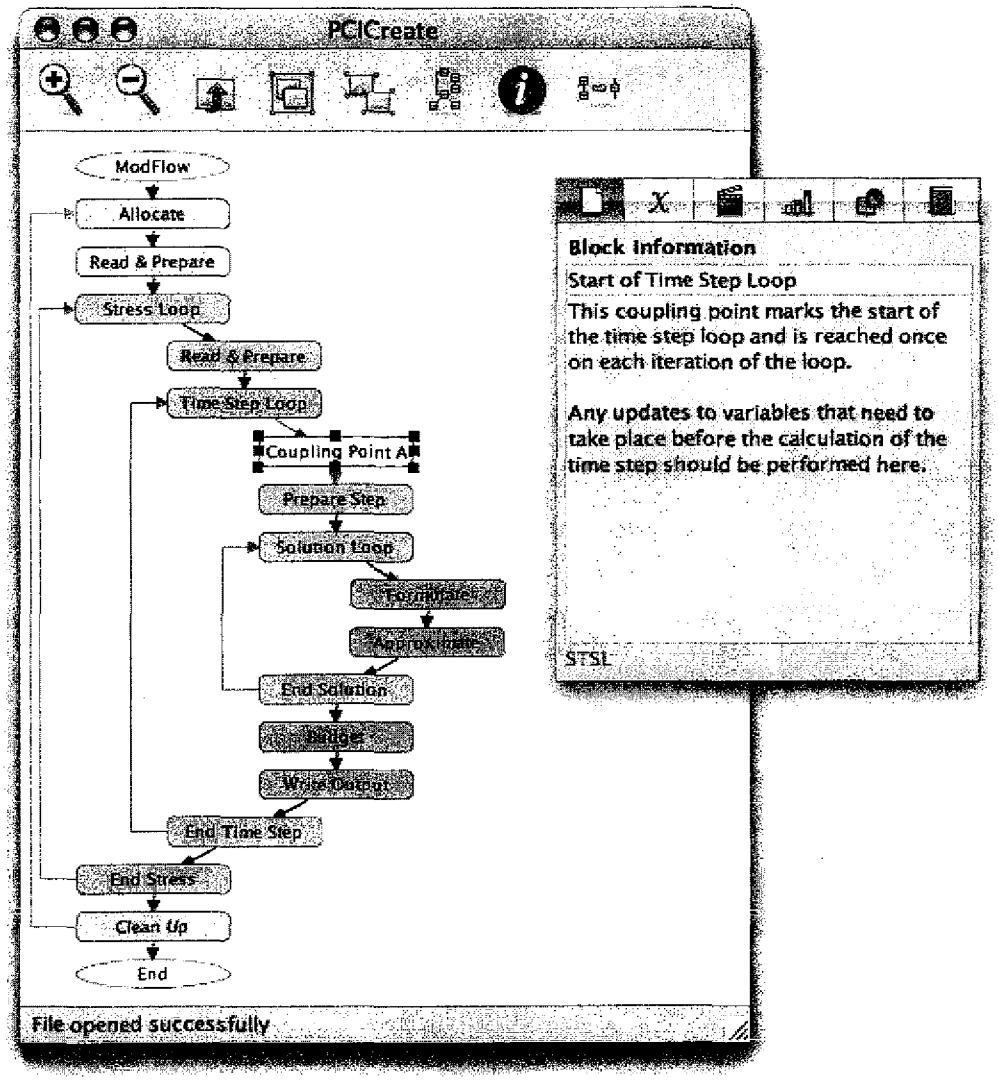

Figure 16. Information about blocks can be viewed in the inspector.

Figure 17 shows another screenshot of PCICreate in which the variable list of the selected coupling point is shown in the inspector window. The scientist can peruse the list of variables that are accessible at a coupling point, inspecting their properties and descriptions. This information about the variables was added by the scientist in the final step of the PCI creation process.

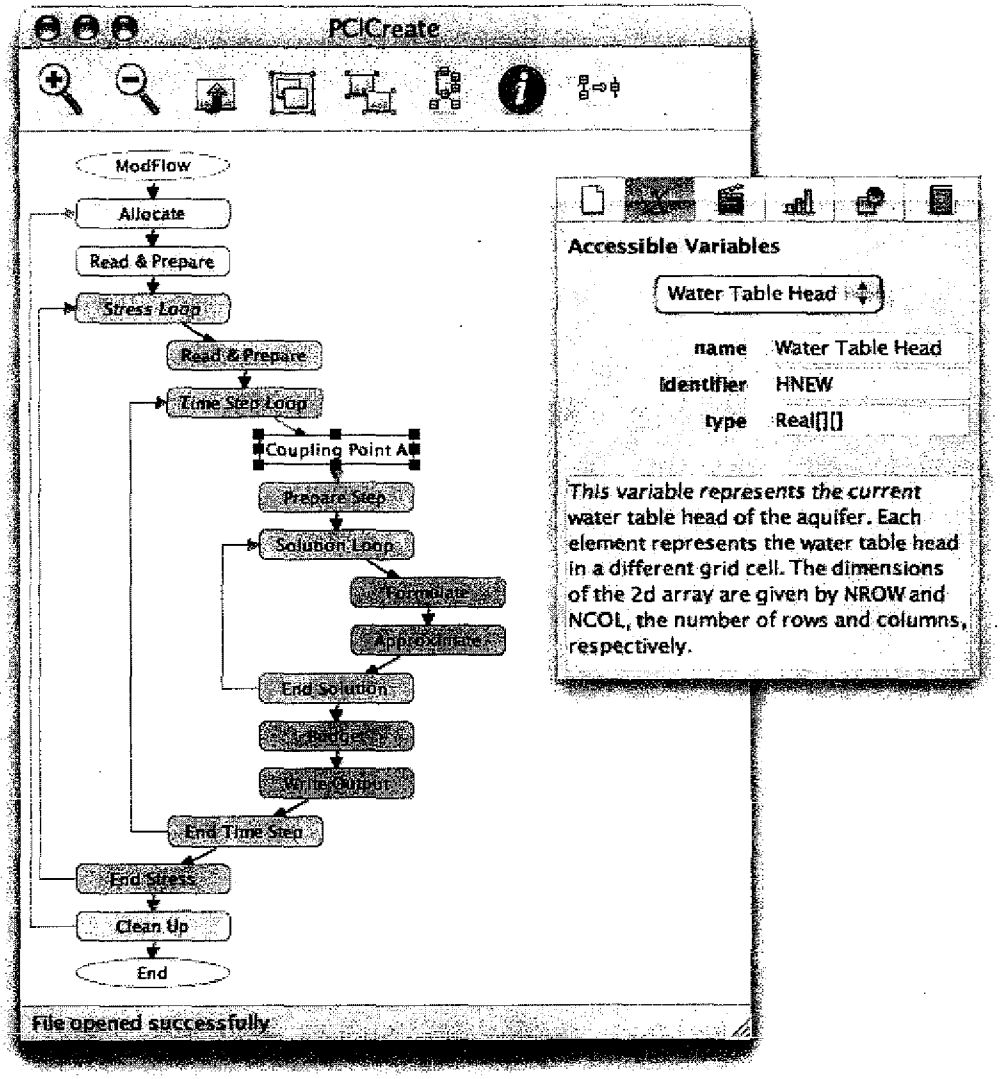

Figure 17. The inspector window shows the accessible variables.

We investigated the use of templates of common model structures as a way of standardizing the overall shape of the PCI's graph, but found the structure of simplified flow graphs to vary too much. If all PCIs shared the same overall shape (or a small set of them), they would be easier to understand and compare to each other. Further research on this topic could be beneficial.

How PCIs are Created

This section steps through the process of how PCIs are created. An overview of the process is shown in Figure 18.

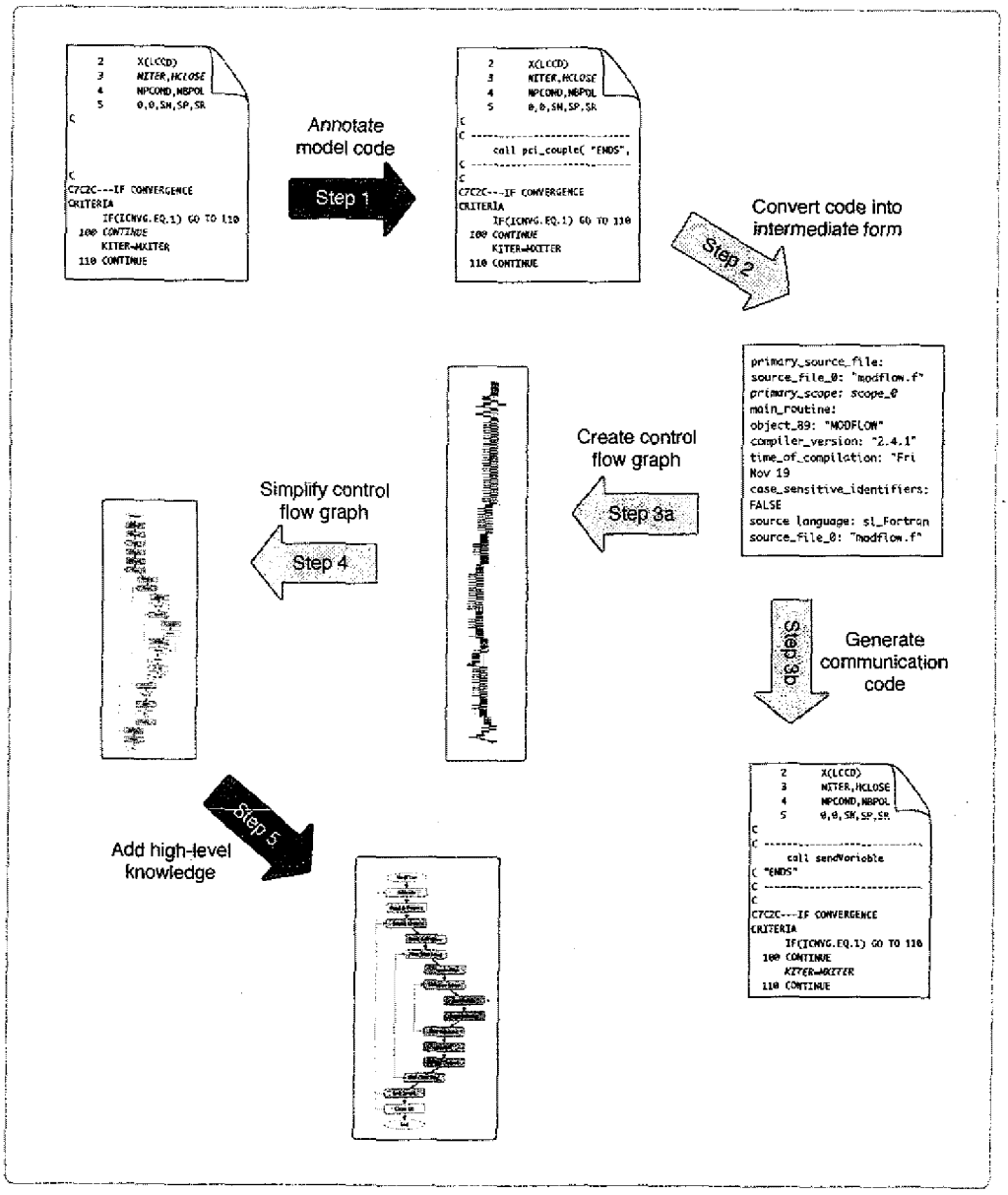

Figure 18. The process of creating a PCI.

Each arrow represents a step in the process of creating a PCI. The dark arrows are steps that are performed by the scientist, and the light arrows are steps per-

formed automatically by PCICreate. In Step 1 the scientist annotates the model code, indicating which state variables should be accessible, and at what points in the source code. In Step 2, the annotated model code is automatically converted into an intermediary (analyzable) form, from which a complete control flow graph is derived (Step 3a) and from which the original model code is instrumented (Step 3b). In Step 4 the complete graph is reduced around the coupling points. The scientist then customizes the simplified graph and incorporates any necessary domain-level information in Step 5. PCI creation is explained in the next section, and the instrumentation step in the following section.

Creating the PCI

The first step in creating a PCI is to identify the state variables of the model, and specify where they have meaningful values in the model code. Since it is not possible to automatically identify the state variables of a model code in general, the model author or someone familiar with the source code annotates it, marking potential interaction points and state variables that can be read or modified at those points. Figure 19 shows an example of an annotation.

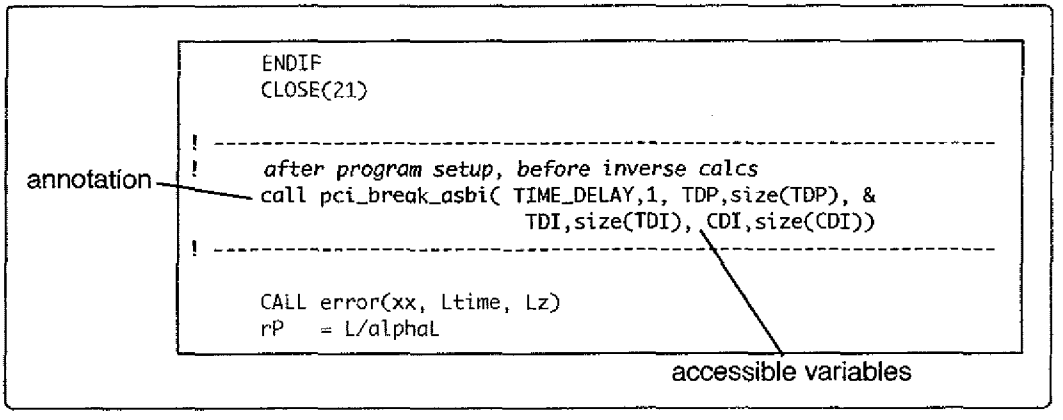

Figure 19. An example of an annotation.

The annotation in the figure indicates that a coupling point should be created at the specified point in the model code, uniquely identified by the name *asbi*, and that the four variables *time_delay*, *tdp*, *tdi*, and *cdi* should be accessible at the

coupling point. These variables are said to be *local* to that coupling point. Each variable in the annotation is followed by a number that indicates the number of elements in the variable (for scalars, the number of elements is 1). The difference between annotations and coupling points is that annotations are program statements added to a model code that indicate that a coupling point should be added to the model's PCI. Annotations take the form of function calls because functions are a familiar programming construct to scientists, and they simplify our implementation. The actual function does not exist at the time of annotation, it serves only as a placeholder and the function is generated automatically in Step 3b. Alternatively, annotations could have the form of compiler directives as done in OpenMP (Dagum and Menon 1998).

Coupling points are either *breaking* or *inline*. If a coupling point is a breaking coupling point (indicated by prefixing the annotation with "pci_break"), then the coupling point will be preserved in the PCI as a block. If a coupling point is an inline coupling point (indicated by prefixing the annotation with "pci_inline"), then the coupling point is not preserved in the PCI and the variables accessible at the point are associated with the block in the graph that represents the section of source code in which the annotation lies. Inline coupling points are used in cases where the accessible locations of several variables are scattered throughout a section of source code, and creating a separate coupling point block in the graph for each location would complicate the graph unnecessarily. Guidelines for annotating model codes is given in Appendix B.

After the model code is annotated, it is imported into PCICreate via the *New* menu item, and is then translated (Step 2 in Figure 18) from its source language into a structured intermediate form (called a program database file) using the Program Database Toolkit (PDT) (Lindlan et al. 2000). PDT supports C, C++, and Fortran. The intermediate form is then parsed by PCICreate in Step 3 to generate a complete control flow graph. With one block in the graph for each

statement in the model code, these graphs are generally far too large to be comprehensible. To reduce them (Step 4 in Figure 18) we use a modified form of the *graph reductions used in interval analysis (Aho and Ullman 1972). The interval* analysis algorithm reduces a graph by collapsing adjacent blocks. It inspects only the edges of the blocks and treats all blocks the same. Our algorithm additionally inspects the type of each block (control statements such as loops, jumps, and conditionals, and annotation statements), enabling us to preserve annotations and the control structures surrounding them in the reduced graph.

The interval analysis algorithm partitions a flow graph into disjoint intervals (subgraphs). Each interval contains at least one node (we use the term *node* rather than *block* throughout the remainder of this section since this is the terminology used by the algorithm authors), known as the header node, from which all the other nodes in the interval can be reached by following only forward-edges. The algorithm traverses the flow graph incrementally adding nodes to each interval. When a node is encountered that has an in-edge that originates from a node that is not already in the interval (and if that node is not a header node), then a new interval is created for that node (in which it is the header node) and the process continues. After the intervals have been partitioned, each interval is added as a node in the *derived graph*, which is the final reduced graph. An example of how interval analysis collapses a subgraph is shown in Figure 20.

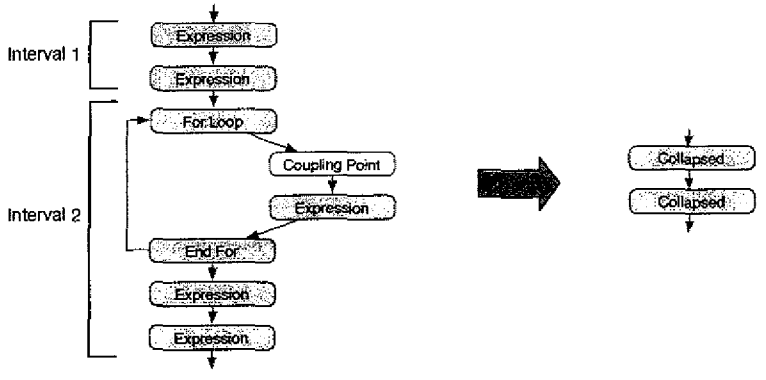

Figure 20. How interval analysis collapses a subgraph.

The original graph is shown on the left of the figure, and the derived graph on the right. The first two "Expression" nodes are added to interval 1, but since the "For Loop" node has an incoming edge that originates from a node outside this interval, it cannot be added to the interval. Rather, a new interval is started, interval 2, to which the "For Loop" node, and the remaining nodes are added.

The original algorithm was modified for our purposes with two additional rules. The first rule is that nodes in the graph that correspond to breaking coupling points are not collapsed (note though, that inline coupling points are collapsed, so there may be several inline coupling points within a collapsed node). The second rule is that control structures, such as loops and conditionals, whose bodies contain nodes that correspond to breaking coupling points, are not collapsed either. This is because these control structures are closely related to the coupling point since they dictate when the coupling point is reached during execution of the model. The algorithm preserves loop constructs such as *for*, *while*, and *do* loops, and it also preserves *goto* statements that span coupling points since they are sometimes used as loops (both forward and backward jumps are preserved). An example of how our algorithm collapses a subgraph is shown in Figure 21. Notice how the loop nodes are preserved since there is a coupling point within the body of the loop.

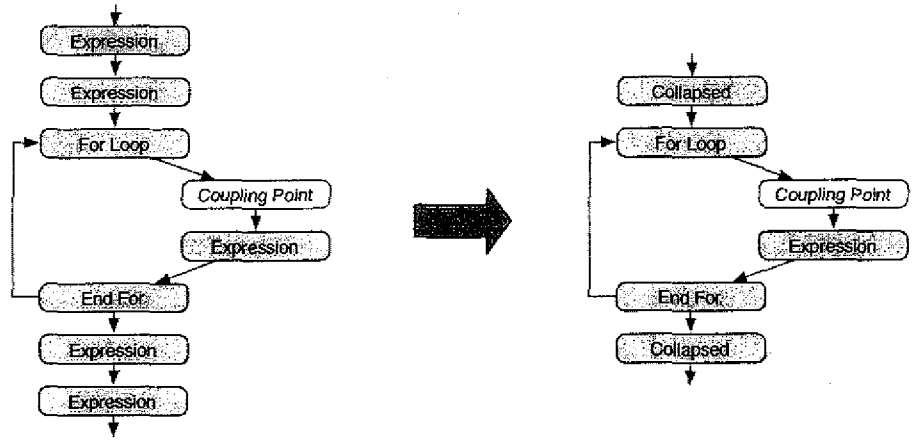

Figure 21. How our algorithm collapses a subgraph.

By changing the placement of the annotations in the model code, a different reduced graph will result. Figure 22 shows how the same source code used in Figure 21 with annotations at different locations will result in a different reduced graph. In this example, the loop is collapsed because there are no coupling points within its body.

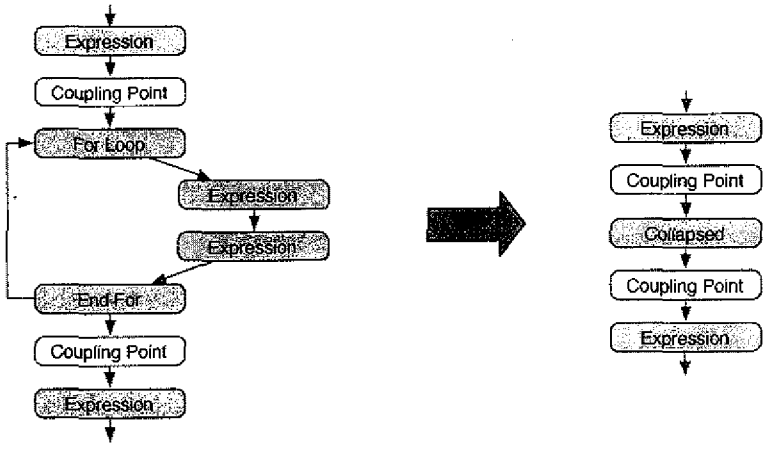

Figure 22. How a different placement of annotations affects the reduced graph.

This reduction process significantly reduces the number of nodes of a graph. In the case of the ModFlow model, the complete CFG consisted of 10,158 nodes, while the reduced graph consisted of only 35 nodes. The reduced size is only 0.50% of the original size. Table 3 shows the reduction ratios for several models. In the figure, the number of nodes in the reduced CFG is the number of nodes in the PCI after automatic reduction before any user changes.

Table 3. Reduction ratios for various hydrological models.

Model	Number of nodes in complete CFG	Number of annotations	Number of nodes in reduced CFG	Percent of original size
ModFlow	10,158	16	35	0.50%
DAFlow	672	10	25	5.21%
STAMMT-L	2,159	9	12	0.97%
TopModel	431	15	35	11.6%
SHAW	5,758	11	57	1.18%

The number of annotations placed in a model code will directly affect the number of nodes in the reduced control flow graph. The fewer the annotations, the smaller the reduced graph (a model with no annotations would be reduced to a single node). A precise relationship between the number of annotations and the size of the reduced graph is not possible since annotations located within nested control structures will result in a larger reduced graph (due to the preservation of the control structures) than annotations located outside any control structure.

After the reduced graph has been created, the PCI creator must edit it to enhance readability and to integrate domain-level information (Step 5). Some of the editing features include collapsing, labeling, and coloring blocks, grouping blocks, viewing the underlying source code associated with a block, adding descriptions to blocks and variables, and arranging the visual placement of the blocks. The final PCI is then stored and reused thereafter. An explanation of how the model code is automatically instrumented by PCICreate (Step 3b) is presented next.

Instrumenting the Model Codes

The annotations indicate that variables are accessible at various points throughout a model code, but they do not enable the model to actually send and receive the values of these variables. The function that is called by each annotation must be defined, which requires additional source code to be generated and added to the model codes. Our original design called for *late instrumentation* in which the model codes are instrumented after the coupling description is specified, and the instrumentation is coupling-specific (the models could only communicate with each other, as specified in a particular coupling description). Our final design though, uses *early instrumentation* in which a model code is instrumented immediately after its PCI has been created (Step 3b in Figure 18). The instrumentation is coupling-independent allowing the model to communicate with any other model. These approaches are compared in Figure 23.

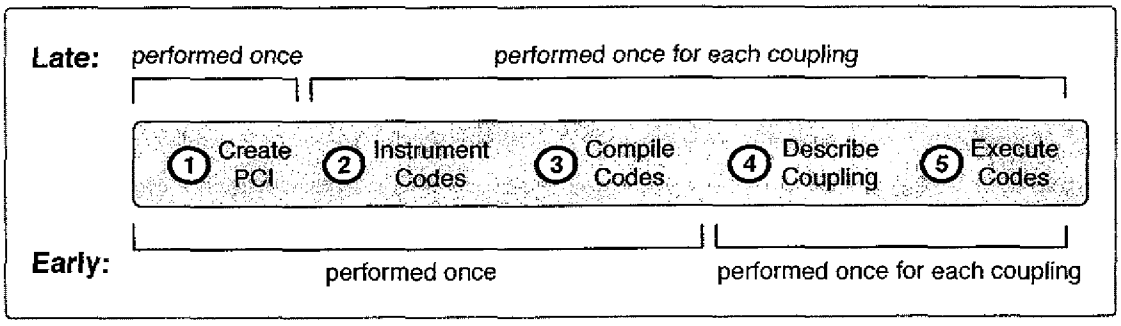

Figure 23. Early vs. late instrumentation of model codes.

As shown in the figure, in the late-instrumentation approach the PCI creation is the only step that is performed once, and the rest of the steps are performed once for each coupling, and the instrumentation is coupling-specific. In the early-instrumentation approach, the PCI creation, along with the model code instrumentation and compilation, are all performed only once. The advantage of the late approach is that the instrumentation itself is simpler since it must only be instrumented to carry out a specific interaction with a specific set of models, rather than the more complex instrumentation that is capable of sending and receiving any variable at any coupling point as in the early approach. Furthermore, coupling-independent instrumentation also requires that each model be told dynamically what to do at each coupling point at runtime, again, complicating the instrumentation. The advantage of the early approach is that it requires less effort on the part of the scientist who is coupling the models. It removes the burden of compiling the model codes after every change to the coupling description. Compilation of model codes can be difficult due to differences in computing platforms and compilers, so avoiding this step allows the scientist to more rapidly execute the coupled model. Since the scientist will likely develop the coupling iteratively, incrementally making it more complex and testing it, a significant amount of time can be saved. This iterative development process is particularly important in implementing distributed systems (such as coupled models), due to the difficulty and complexity inherent in their design and use. Early instrumentation also has the advantage that closed source model codes can still be used in couplings

since the scientist does not need to compile the model codes him/herself. The model author can create the PCI and coupling-ready executable for their closed source model, and release them to the public without the source code. Although the early-instrumentation approach requires more complex instrumentation, it does not limit the utility of our approach to model coupling, and facilitates the scientist's task of quickly prototyping coupled models.

In Step 3b of the PCI creation process shown in Figure 18, coupling-independent communication source code is added to the model source code, enabling the model to send and receive the value of any variable at any coupling point. Which variables are sent and received in a particular coupling is described by a *script*, a list of put and get events. The combination of coupling-independent instrumentation and coupling-specific scripts, as illustrated in Figure 24, allows instrumented model codes to be reused, while at the same time allowing the behavior of the model to differ in different couplings.

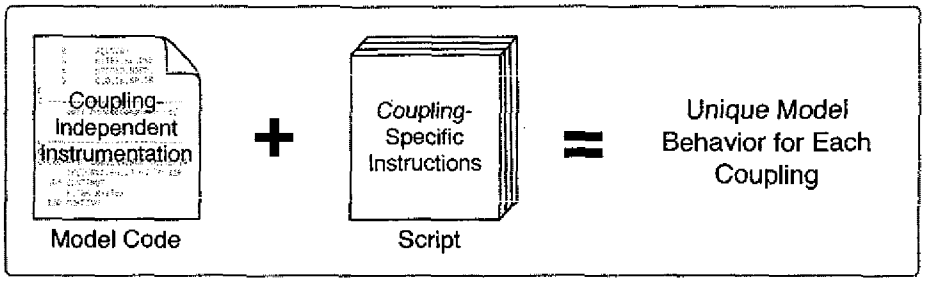

Figure 24. Scripts describe coupling-specific behavior.

Scripts are created automatically by PCICreate and are used behind-the-scenes in the execution of coupled models. They are described in detail in Chapter 6. The model code instrumentation, generated by PCICreate, enables a model code to receive a script and carry out the events described by it. This instrumentation process is explained next.

To instrument a model code, PCICreate begins by adding a small amount of communication source code at the very start of the annotated model code that

enables the model to receive a script. It then automatically generates a custom *accessor subroutine* for each annotation. These accessor subroutines are capable of sending and receiving any variable at a given coupling point, as directed by a script. Since different variables may be accessible at different coupling points, each accessor subroutine is different, and specific to the variables available at a particular coupling point. An example is shown in Figure 25.

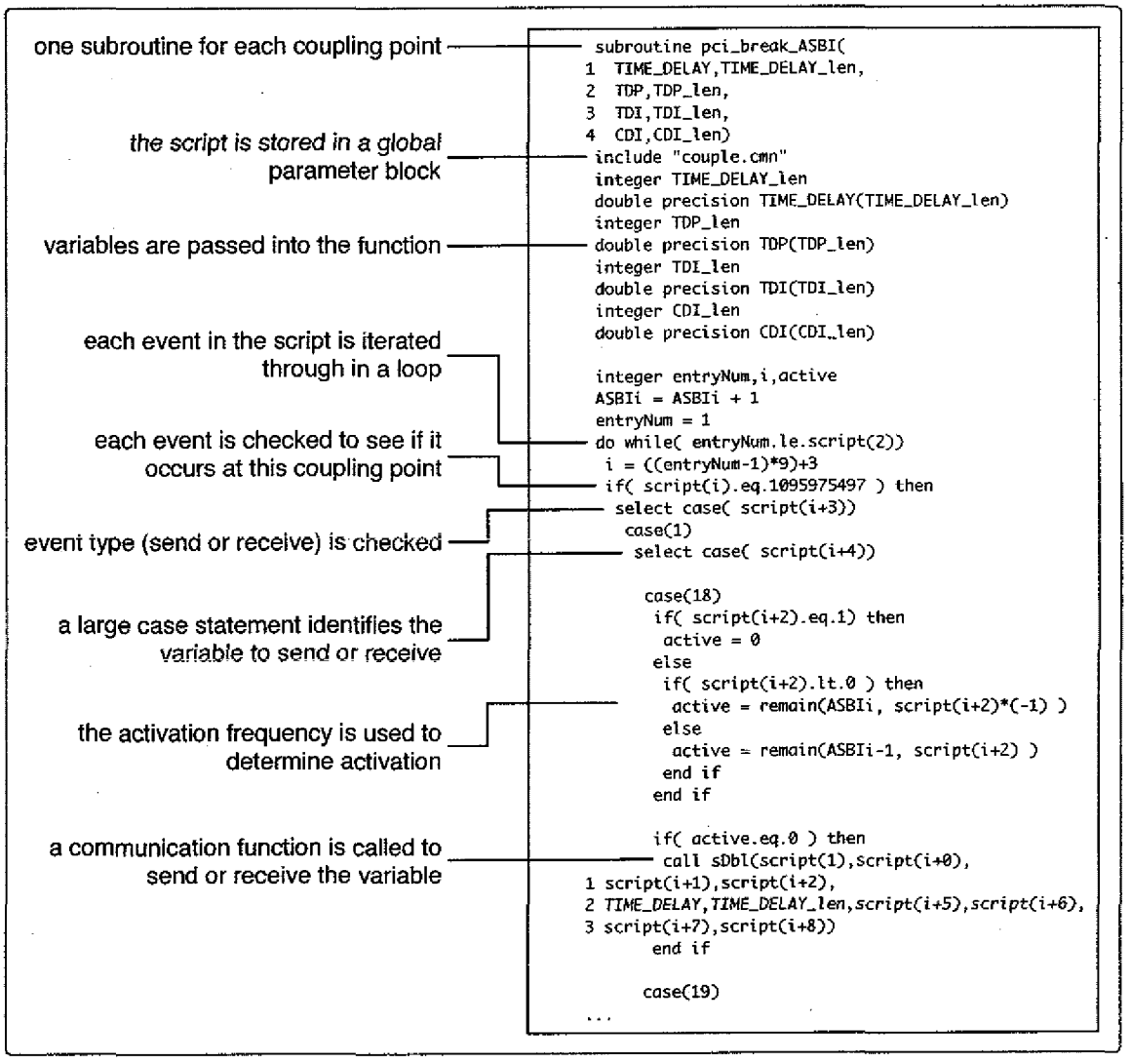

Figure 25. Source code of the generated accessor subroutine for the annotation shown in Figure 19.

The accessible variables are passed into the function (see the annotation example in Figure 19), the script itself is stored in a common block (couple.cmn). The function consists of a large do loop that iterates through each event in the script. The condition in the loop checks to see if the event is for this coupling point, and if so, switches the appropriate event execution.

The overhead of the generated subroutines, in terms of lines of source code, grows linearly with the number of variables at a coupling point. The number of lines of source code for a generated function is 10 lines, plus 25 lines per variable. In the complete function shown in the figure, which has 4 variables, there are 110 lines of source code. The overhead of the generated subroutines, in terms of execution speed, is minimal compared to the overhead involved in sending and receiving data between instances. Each iteration of the script event loop performs a single switch statement, and the number of events in a script will generally be small (under 50). After the model code has been instrumented, it is compiled by the scientist to create a *coupling-ready* executable. The PCI and executable can then be distributed and reused by scientists wishing to use the model in a coupling.

The communication subroutine call shown in the figure, *sDbl*, can send any double precision variable. There are analogous functions for sending and receiving real, integer, logical, and character types as well. Differences in the number of bytes used to store variables (long integer vs. integer, and double precision vs. real) are kept track of in the script and that information is used by the communication subroutines. The communication subroutines that are called from the generated coupling point functions are part of a custom communication library that we developed. The library is written in C and provides access to standard TCP sockets. The functions are essentially wrappers around TCP's *send* and *recv* functions, that add additional resiliency and error handling specific to our runtime environment. The library functions do not interpret the values that are

sent and received: all parameters to the functions are unsigned byte arrays, allowing for any data type of any size and shape to be used.

The library is written in C, so C-Fortran interoperability techniques are necessary to allow Fortran programs to use it. If the object file format used by both the C and Fortran compilers is the same (taking care that the name mangling is compatible), then the library and the model code object files can simply be linked together to create the final executable. Some compilers support C-Fortran interoperability through use of proprietary compiler directives. An additional issue regarding the exchange of data between Fortran and C is that multi-dimensional arrays are internally stored in C/C++ in row major order, while Fortran stores them in column major order. If arrays are communicated between models written in different languages, then the arrays must be translated during communication, which can either be performed at the application level (via transformations within the coupling framework), or at the instrumentation level. Such transformations can be supported by incorporating existing work in this area (Rasmussen et al. 2001).

It would be possible to substitute any communication library in place of our custom library that provides the same point-to-point style communication. Since the communication library is based on TCP sockets, coupled models can be executed in a distributed fashion where each model is executed on a machine at a different geographic location. This is consistent with the emerging Grid research, in which management of massively distributed computational resources are organized and coordinated to support distributed simulation (Armstrong et al. 2005). Exchanging data across different computing platforms introduces an additional issue that must be handled, byte-ordering. For example, Motorola processors use big endian byte ordering, while Intel processors typically use little endian byte ordering. Any data exchanged between models running on platforms

based on these different processors must be reordered at the byte level accordingly.

Is There More To Coupling Potential?

The basic elements of coupling potential presented in this section are clearly necessary for creating coupled models, but are they sufficient? Are there additional elements related to the model, its structure, or its variables that are also necessary? To investigate this question we conducted a study of coupled models in which we stepped through the process of designing a series of couplings based on PCIs (Bulatewicz and Cuny 2005). This empirical approach showed us what worked well in our initial PCI design, and more importantly, showed us what additional model characteristics must be described in the PCI.

The Coupled Model Study

The purpose of this investigation was to identify the common characteristics of models that affect coupling compatibility and to assess the degree of compatibility of model pairs based on those characteristics.

Methodology

We studied a set of 14 hydrological models and the design of the 91 pairwise couplings between them. The models, which varied in complexity, are serial programs written in Fortran and are representative of the different kinds of systems commonly modeled in hydrology: groundwater (G), surfacewater (S), rainfall-runoff (R), receiving water (W), and field (F) systems, and are listed in Table 4. For each pairing of these models, we created the appropriate PCIs (using an initial design that described the basic elements presented above) and used them to design a coupling between the models (a PCI for each is given in Appendix A). The design of each coupled model followed steps 2-5 of the model coupling process described in Chapter 3. Step 1, choosing the models, was not

applicable because the models of interest were already collected; Step 6, instrumenting the model codes, was not applicable because we were interested only in the design of the couplings, not executing them. What we learned in our study from the remaining four steps is described in the next section.

Table 4. Models used in the coupling study.

Model	Description	Kind	Reference
BioMOC	Groundwater-flow and transport	G	Essaid and Bekins 1997
Branch	Surfacewater-flow	S	Schaffranek, Baltzer, and Goldberg 1981
DAFlow	Surfacewater-flow	S	Jobson 1989
FourPt	Surfacewater-flow	S	DeLong, Thompson, and Lee 1997
GLEAMS	Soil chemistry and runoff	F	Leonard, Knisel, and Still 1987
ModFlow	Groundwater-flow	G	McDonald and Harbaugh 1988
OTIS	Surfacewater transport	S	Runkel 1998
SHAW	Soil chemistry and runoff	F	Flerchinger 2000
STAMMT-L	Surfacewater transport	S	Haggerty and Reeves 2003
SWAT	Rainfall-runoff and transport	R	Neitsch et al. 2001
SWMM	Rainfall-runoff	R	Huber and Dickinson 1988
TopModel	Rainfall-runoff	R	Beven 1997
UEB	Snow-melt	R	Tarboton and Luce 1996
WASP	Receiving water	W	Ambrose, Wool, and Martin 1993

The physical systems simulated by the models in our set influence each other in many ways, but since water (represented as a height, flow rate, volume, etc.) is the physical quantity common to all the systems, we focused on the water flux interaction between the systems. Most water flux interactions can be classified into the nine kinds illustrated in Figure 26, many of which have been studied previously with coupled models (Johnston et al. 2003; Jobson and Harbaugh 1999; Ross et al. 2004; etc.). As indicated by the arrow directions in the figure, these interactions can be unidirectional or bidirectional. All of them are continuous over time.

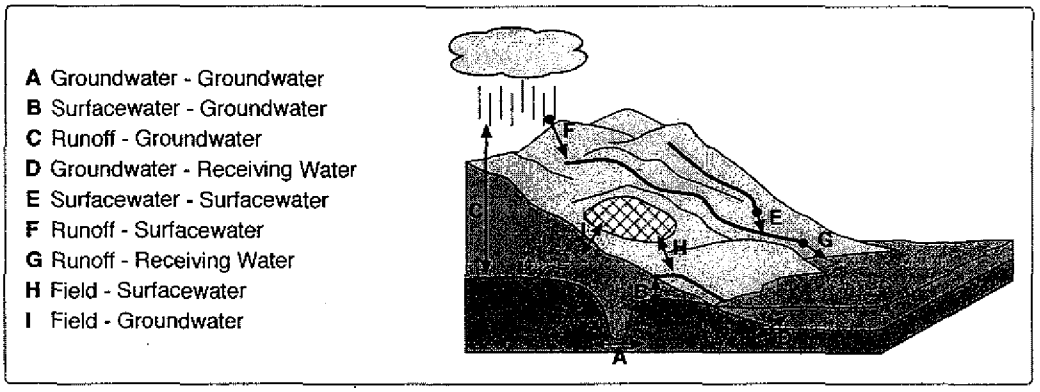

Figure 26. Interactions between hydrological systems.

Results

Since our intent was to study the water flux between these physical systems, we first identified the coupling surface (Step 2), at the domain-level, across which the models would exchange water. Each arrow in Figure 26 crosses a coupling surface, and the one used in a particular coupling was dependent upon the physical processes simulated by the two models.

The next step was to identify the state variables in each model that represent the physical quantity of water (Step 3 in the coupling process). This was trivial because the PCI provides a description of each variable, making it easy to identify the variables associated with a particular physical quantity. After identifying the relevant state variables of each model, it was necessary to develop a clear understanding of their syntax and semantics, which varied considerably across our set of models (Step 3). The PCIs included syntactic information about the variables, such as data type and shape, but they did not (yet) describe the semantics. Key to the semantics of this data is its spatial distribution: modeled quantities were distributed as a set of 0d points, vertically along a 1d profile, horizontally along a 1d line, as a set of 2d points arranged on a surface, as a 2d regular or irregular grid, a 2d regular grid cross section, or a 3d regular grid volume. Within a single model, different variables were often distributed in different ways. In addition to the distribution, the spatial scale (field, catchment, basin,

etc.) at which the variables were distributed varied significantly. This flexibility in spatial distribution and scale allows the models to be used at a variety of sites (where the particular spatial configuration used in a model run is dependent upon the study site) but it can complicate the design of the coupling as discussed in Chapter 5. Step 3 showed the importance of the spatial distribution and scale of state variables when relating variables between models. This step was performed once for each model.

After the variables were identified and understood, the locations within the model code where they should be exchanged was determined (Step 4). The PCIs of each model limit access to state variables to locations where those variables are meaningful, thus, it was necessary only to insure that the data exchanges happened at consistent locations, that is, locations in the model codes that represent the same point in simulation time. This was determined by considering both the surrounding control structure (loops, conditionals, etc.) and the time step length (and whether it varies or not), which collectively dictate the set of (simulation) times at which a variable is accessible; this is further discussed in Chapter 5. The length of time steps supported by our models varied considerably and were often restricted to a particular range for convergence, accuracy, or efficiency reasons. The duration of a simulation is important because it dictates the span of (simulation) time during which the state of a physical quantity is accessible, although most models did not limit the duration of a simulation and were capable of simulating very long periods of time (several years). Step 4 showed the importance of the temporal characteristics of models, particularly the length of the time step, and its relationship to the control structure.

After identifying the coupling points at which variables should be exchanged, the qualitative relationship between the variables of each model was determined (Step 5). In this step, we specified the functional relationship between the state variables of each coupling. These functions varied in their complexity

and were customized for each coupling. In some cases, the value of a state variable simply overwrote the value of another, but in most cases, non-trivial calculations were necessary. This step was performed once for each model pair.

Our purpose for conducting this study was to identify the additional elements of coupling potential beyond the basic elements identified in the previous section. We found that the original design of the PCI sufficiently allowed for easy identification of coupling points, state variables, and the control structure around them, but that more information about the meaning of the state variables was necessary in order to correctly relate variables between models. Table 5 summarizes the additional characteristics of state variables that must be described in the PCI.

Table 5. Coupling-relevant variable characteristics identified in the study.

Variable Characteristic	Variation Found in Study
Spatial Distribution	0d, 1d profile, 1d channel, 2d points on a surface, 2d regular grid surface, 2d irregular grid surface, 2d regular grid cross section, 3d regular grid volume
Spatial Scale	field, catchment, basin, etc.
Time Step Properties	length: short, hourly, daily, weekly, monthly; variable or constant

In order to better understand the extent to which differences between models influence how they can be coupled, we compared each pair of models in terms of their coupling potential.

Model Compatibility

We rated the similarity of each pair of models with respect to the elements of coupling potential identified in this chapter. We classified the elements identified above into four dimensions of similarity: space (distribution and scale), time (time step properties), structure (control structure of the model code), and data (physical quantities). Then, for each model pair, we compared the models with

respect to their similarity in each of these four dimensions. Along each dimension, a model pair was rated as either similar or different. With respect to the spatial dimension, two models are considered similar if they both support at least one common spatial distribution and scale, and different otherwise. In the temporal dimension, a model pair is similar if both models support at least one common constant time step length, and different otherwise. A model pair is similar along the structural dimension if the models possess the same nesting of loop kinds (time step loop, spatial loop, solution loop, etc.), and different otherwise. With respect to data, two models are different if one simulates only water at the surface, and the other only water below the surface, and similar otherwise. A summary of the model similarity for all of the pairings is shown in Figure 27.

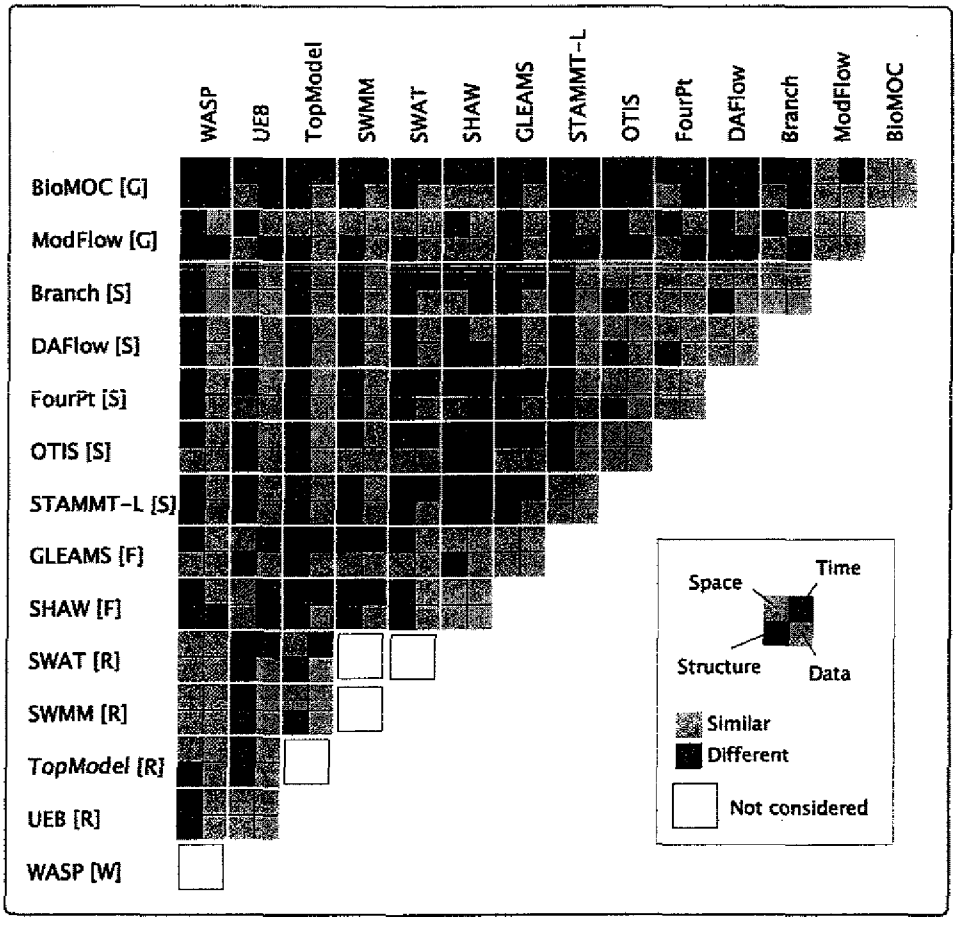

Figure 27. Summary of similarities and differences between the models.

In general, the more similar the models, the easier it will be to couple them. The models in the figure are grouped according to their simulated processes (e.g. ModFlow and BioMOC are both groundwater models, so they're listed together). This ordering reveals that models of the same phenomena are often similar with respect to their coupling potential. This suggests that once one of them has been used in a coupling, swapping it for another (that may be more appropriate for a particular site, for example) will generally be straightforward. Couplings that lacked a sensible interaction between the modeled systems were not evaluated and are indicated in the diagram by white squares. These were mostly couplings between runoff models, which at large catchment scales do not interact with other catchments.

This is a comparison of the models' base compatibility. As an example, consider the compatibility of ModFlow and TopModel (discussed in Chapter 7). Although the figure indicates that the models differ only along the structural dimension, this is not the only incompatibility. While some modeled quantities are spatially distributed in the same way in both models (hence, rated as similar in the figure), some quantities, such as ModFlow's groundwater height and TopModel's groundwater height, are not spatially distributed in the same way and present an additional incompatibility between these models. Similarly, the application of a model to a specific site may introduce other incompatibilities, as an example, inputs for both models may not be available for the same period of time, such as long term precipitation records vs. short term stream-flow records.

The figure indicates wide dissimilarity along the structural dimension, with only 30% of the model pairs marked as similar. Although nearly all the models possessed a time-stepped loop, only half possessed a central solution loop, resulting in the low similarity. Furthermore, DAFlow and TopModel were unique in their inclusion of spatial loops, and STAMMT-L was unique because it possessed no time-stepped loop, both of which further reduced the overall structural similar-

ity of the models in the set. With respect to time, 60% of the pairs were found to be similar, which is expected due to the flexibility in the time step lengths supported by the models. With respect to the similarity of the water quantities modeled, 70% of the pairs were found to be similar. The high similarity is due to the variety of water quantities included in each model. Of the four dimensions, the spatial similarity was the lowest, with only 20% of the models marked as similar. This can be explained by the high variability in the spatial characteristics of the models. Over the four dimensions of characteristics, though, the models were more similar than dissimilar, suggesting that there are many opportunities for coupling models within the domain of hydrology. The dissimilarities though, indicate that incompatibilities between models are common, and that techniques are needed to resolve these differences. This is discussed in the next chapter.

Summary

Through our work in early case studies we were able to identify the basic elements of coupling potential and then design a representation, the PCI, that succinctly describes it. Our coupled model study revealed what was lacking in the original design: detailed information about the meaning of the state variables. Our look at model compatibility showed us that differences between models are common (in our set of models), and must therefore be resolved when designing a coupling. The next chapter presents a language for describing coupled models in terms of their PCIs and explains how the language supports resolving incompatibilities between models.

CHAPTER V

DESCRIBING COUPLED MODELS

Introduction

*This chapter shows how the behavior of a coupled model can be de*scribed in terms of the models' Potential Coupling Interfaces (PCI). We first present an overview of the coupling language and the environment in which it is used. We then discuss how incompatibilities can be resolved and present a pair of examples that demonstrate the process of creating a coupled model.

Overview

Scientists describe the interactions between models in the coupling environment (CE), provided by our PCICouple application. In this way, PCICreate is used in the creation of PCIs, and PCICouple is used to describe and execute couplings. The inspector window is used to inspect different aspects of a PCI, just as in PCICreate, but in PCICouple it is also includes information about couplings as well. A coupling specification begins with the PCIs for one or more models that are to be coupled. Figure 28 shows an example of a coupling description within PCICouple, with two PCIs, each for a different model. The environment incorporates the Coupling Description Language (CDL) which consists of a set of actions that operate on the accessible variables at coupling points. To describe a coupled model is to specify a set of actions, called an action list, for

each of the relevant coupling points. On the left of the figure, the action list is displayed for Coupling Point A which has been expanded by the user. During execution of the coupled model, when a model reaches a coupling point, the actions in its action list are carried out. There are three kinds of actions: Send, Update, and Store. Collectively, they allow the values of a model's variables to be changed based on the values of other variables from that model, or from other, coupled models. Blocks for which an action list has not been specified appear with rounded corners, and those with action lists appear with sharp corners with the actions listed in the block.

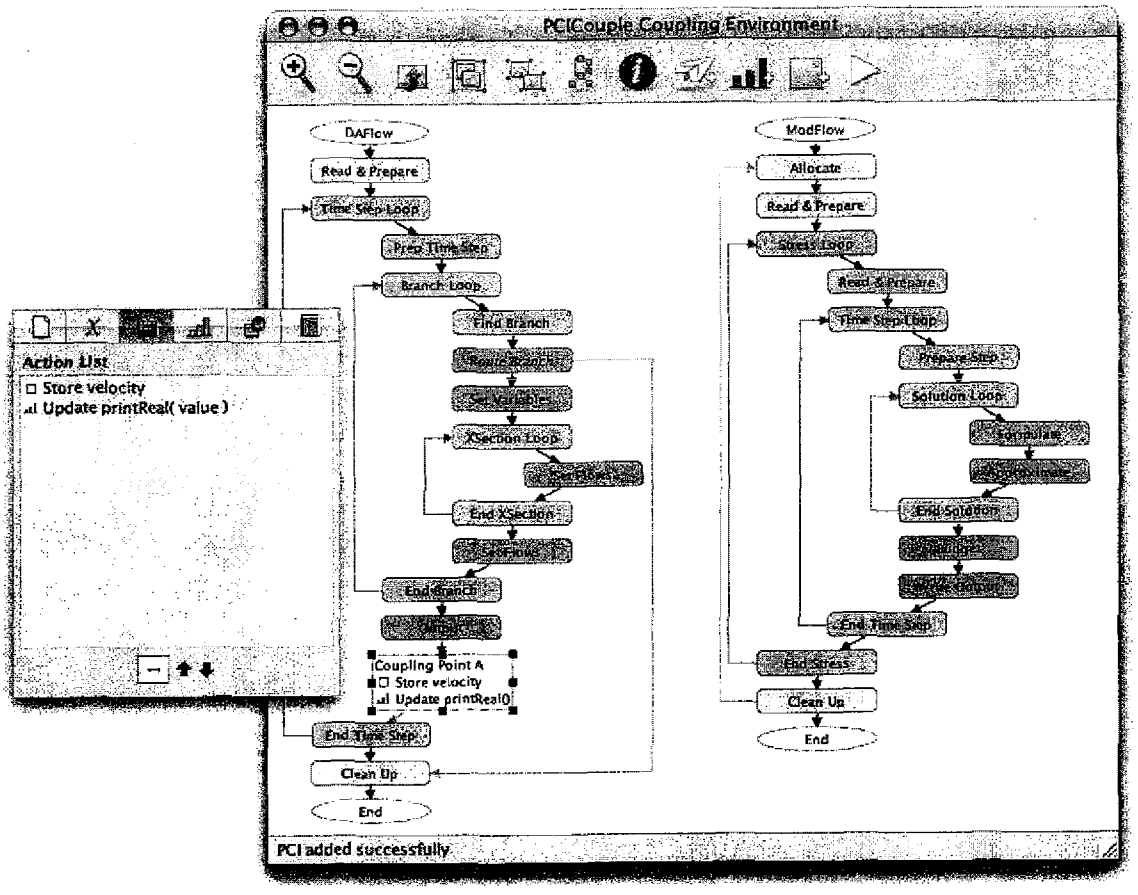

Figure 28. The coupling environment of PCICouple.

Most of the buttons along the top of the window are identical to those in PCICreate, but the collapse button is replaced by three action buttons (left to

right): *send*, *update*, and *store*, and the *execute* button. Actions are added to the action list of a coupling point by clicking the block to select it, and then clicking the appropriate action button. The menu provides options to save and load a coupling description, add PCIs to a coupling description, save a description of the coupling in HTML, save an image of the coupling description, and execute the coupled model.

The Actions

In a coupled model, the values of variables within one model change based on the values of variables within other models. The actions of the CDL provide a means to accomplishing this. We explain each of the three actions next.

The Send Action

The Send Action allows the value of a variable at one coupling point to be used at another coupling point. Send Actions are explicitly depicted in the coupling environment by a labeled, dashed line between the source and destination coupling points in the PCIs, as shown in Figure 29. Solid lines indicate the flow of control, and dashed lines indicate the flow of data. In the figure, the value of variable *hnew* is sent from Coupling Point A in the model on the left, to Coupling Point B in the model on the right. The explicit depiction of Send Actions in the environment makes the communication between the models clear. This is analogous to parallel programming languages such as ZPL (Chamberlain et al. 2000) which make communication explicit in the source code, in contrast to other approaches such as OpenMP where communication is implicitly specified in the source code.

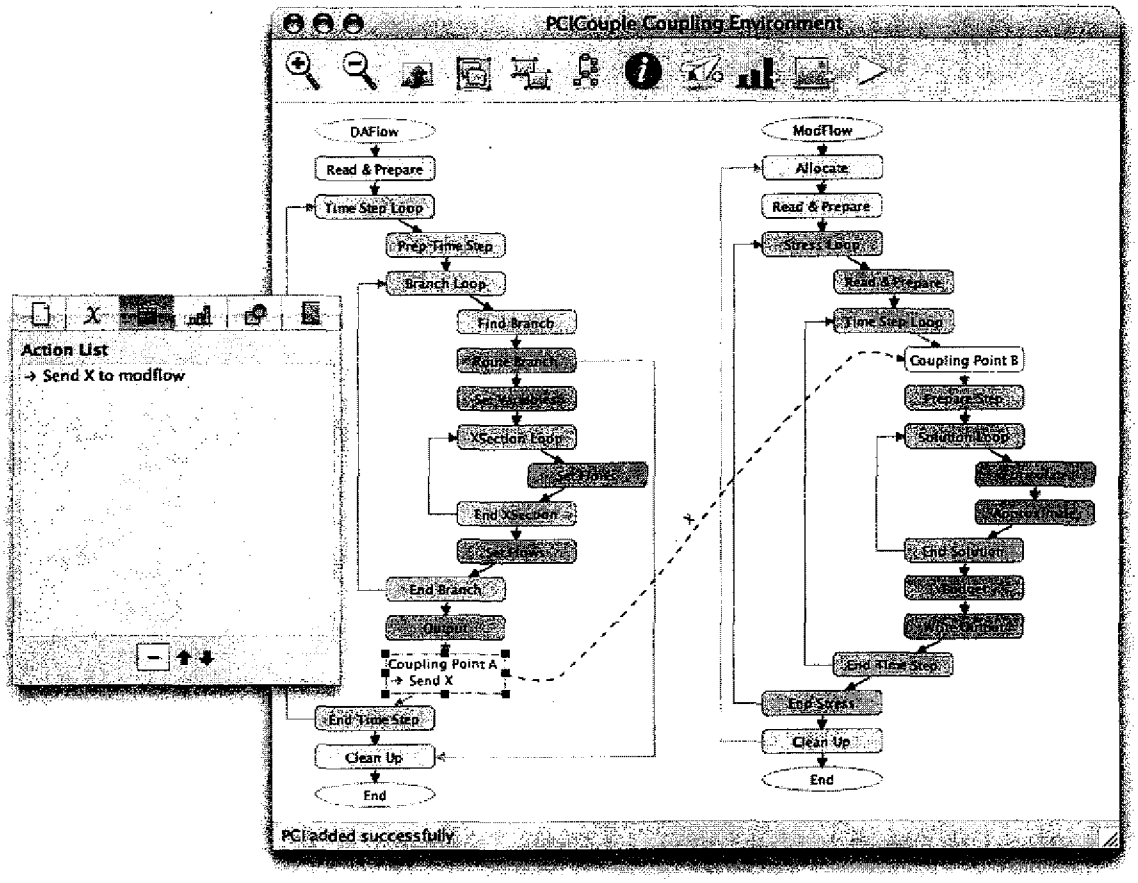

Figure 29. The explicit depiction of Send Actions.

Variable values can be sent from one instance of a model to an instance of different model, or to a different instance of the same model. In the latter case, the depiction within the coupling environment is a dashed line between coupling points of the same PCI, as shown in Figure 30.

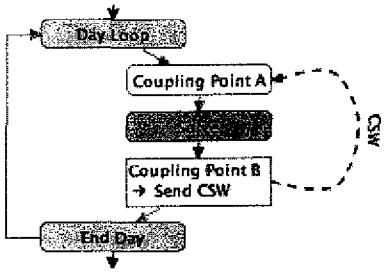

Figure 30. An intra-model Send Action.

When a scientist adds a Send Action to a coupling specification, the sent variable appears in the list of accessible variables at the destination coupling point, making it usable by the actions at that point. The variable name is prefixed with the name of the sending model to avoid name conflicts as shown in Figure 31, where the model name "daflow" is prepended to the variable name *X*.

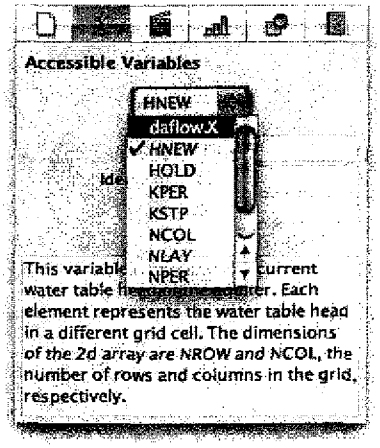

Figure 31. The sent variable X is now accessible at Coupling Point B.

Sent variables cannot be changed or sent again from the destination coupling point, so the value is only usable by Store Actions and as a read-only variable in Update Actions (the ability to re-send variable values could be implemented, but this was not necessary for our case studies). Each kind of action has a set of properties that are set by the scientist when an action is added to the action list of a coupling point. The properties of the Send Action are variable, frequency, and mapping, as shown in Figure 32. The variable property is the name of the variable to send, which is selected from the list of accessible variables at the coupling point. The frequency property indicates the *activation frequency* of the action, which is how often the action is performed. Normally, the actions in a coupling point's action list are carried out each time execution reaches that point. In some cases though, in order to match the execution rates of different models, the interaction needs to occur at a lesser frequency. This is the purpose of the activation frequency property. It describes how often the action should be performed.

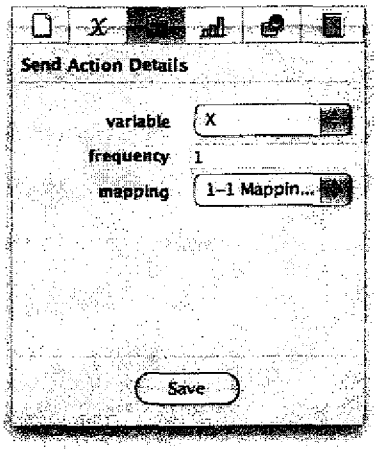

Figure 32. The properties of the Send Action as they appear in the inspector.

For example, an activation frequency of 1 indicates that the action should take place every time the coupling point is reached, while a frequency of 4 indicates that the action should take place every 4th time the coupling point is reached. More general activation mechanisms are possible, but this simple use of frequency has been sufficient for our case studies. The mapping property is the name of the data mapping that should be used for this action. It describes which individual instances should send the variable, and which should receive it. Data mappings are discussed in detail following the presentation of the actions.

The Update Action

The Update Action is used to change the values of variables according to an *update function*, that is, a self-contained (stateless) subroutine that uses variables as arguments. When executed, the update function updates the value of one or more of its arguments. The coupling environment provides a collection of common functions (assign, sum, average, etc.) and the scientist can add their own, as explained in Appendix D. A function is usually written in the same programming language as the model within which it is used. The source code for the built-in *assignReal* function is shown in Figure 33.

```
subroutine assignReal(instanceID,dest,source)
   integer instanceID
   real    dest,source

   dest = source
end
```

Figure 33. Source code for the built-in assignReal function.

Although the function in the figure is very simple, arbitrarily complex computations can also be implemented as needed. The scientist can specify how a set of variables are transformed based on the values of other variables.

A custom language for specifying update functions was not considered because the results of the coupled model study showed that data must typically be transformed in non-trivial ways, requiring the full expressive power of a general purpose programming language. A single, common programming language is not used for update functions because our experience working with scientists showed us that it would be impractical to ask the scientist to learn a new programming language. The properties of the Update Action are function, frequency, and the argument assignments, as shown in Figure 34. The scientist chooses the update function from the list of available functions, and assigns a variable to each of its arguments, chosen from a list of accessible variables.

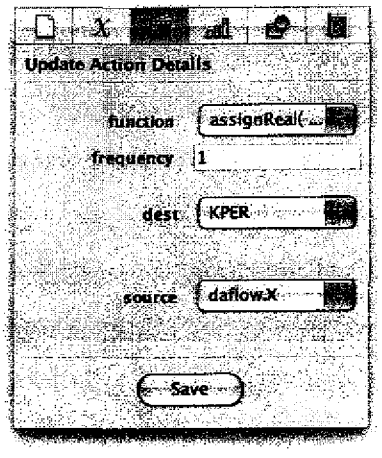

Figure 34. The properties of the Update Action in PCICouple.

If any of the variables used as an argument are sent from another instance, then the Update Action will block until all the parameter values have been received.

The Store Action

Store Actions provide a way to create stored variables, which are new, independent, mutable variables that do not exist in any of the model codes. They are accessible at all the coupling points in a PCI and are usable in Send and Update Actions. Each time a Store Action is activated, the stored variable associated with the action is set to either a (constant) value specified by the scientist, or to the value of a model variable, overwriting the previously stored value (if any). The latter case allows for the value of a variable at one coupling point to be accessed later, at a different coupling point. Stored variables can be used as arguments and updated in multiple Update Actions, providing a way for the result of one Update Action to be used in another Update Action, or in a subsequent activation of the same action. Each instance of each model has its own private data memory in which these variables are stored. The properties of the Store Action are name, set from, and frequency (as well as *variable*, if "Existing" is chosen from the set from menu, and the properties *data type*, *length*, *element size*, and *value* if "Constant" is chosen from the set from menu), as shown in Figure 35.

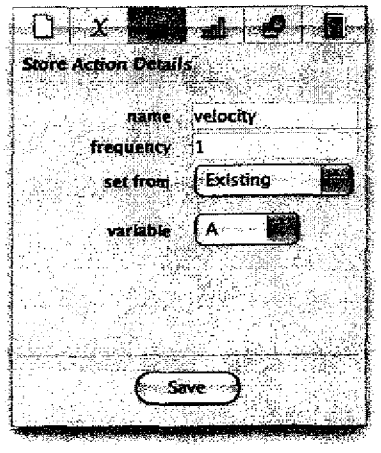

Figure 35. The properties of the Store Action in PCICouple.

The name property is the name of the new stored variable. If its value is to be set to a model variable, then "Existing" is chosen from the set from menu and a variable is selected from a list of model variables and assigned to the variable property. If its value is to be specified by the scientist, then the set from menu is set to "Constant" and the scientist specifies the data type and shape, and the value of the new stored variable.

The actions added to a breaking coupling point are performed in the order specified by the scientist. This allows the actions in an action list to affect subsequent actions in the same list. For example, if an action list consists of two Update Actions that both operate on the same stored variable, then any changes made to the stored variable by the first Update Action would be seen in the second Update Action. However, actions are not ordered if they are added to a block with which two or more inline coupling points are associated. This is because the actions may operate on variables at different inline coupling points, and each inline coupling point may be located at a different place in the control structure of the source code associated with the block.

These actions provide the building blocks for describing coupled models. We may want to couple different models together, or the same model to itself. In the latter case, many instances of the same model code are coupled, where an instance is an executing model process. In cases with multiple instances, showing a PCI for each instance in the CE would make the description cumbersome to create and difficult to understand. For this reason, we separate the description of a coupling into two parts. The interaction between the models is described visually in the CE in terms of the models' PCIs, and the communication between the instances of those models is described textually in data mappings. In this way, only a single PCI is displayed in the CE for each model in a coupling, and it serves as a template that represents the behavior of possibly many instances of that model.

The data mappings are used to describe the communication between model instances. The CE describes the action lists that are to be performed at each coupling point, but which instances communicate with which? Which ones perform which actions? This is described by data mappings, which we explain next.

Sending Data Between Models

Each Send Action requires an associated data mapping that indicates how the value of a variable sent from one model is communicated and transformed before it is received by another model. A simple data mapping is shown in Figure 36 (right), with a partial coupling description (left).

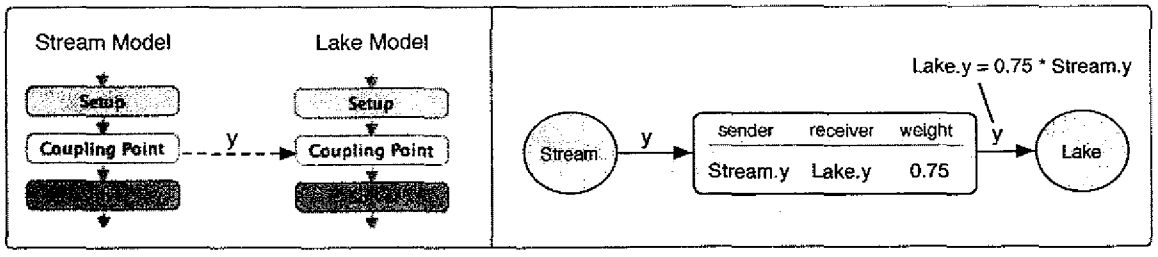

Figure 36. A simple data mapping between two models.

In the figure, the data mapping indicates that the value of the *y* variable sent from the Stream model, should be scaled to 75% before being received by the Lake model. The resulting value of the *y* variable is then available for use in Update and Store Actions at the destination coupling point in the Lake model, and the variable appears in the list of accessible variables at that point. Data mappings describe how one variable is transformed as it is communicated from one model to another by a Send Action. At most one variable can be sent in any single Send Action, and if two variables need to be transformed in the same way, then the specification of the data mapping can be reused. Figure 37 shows two Send Actions, each with its own data mapping, sending the same variable to two different models.

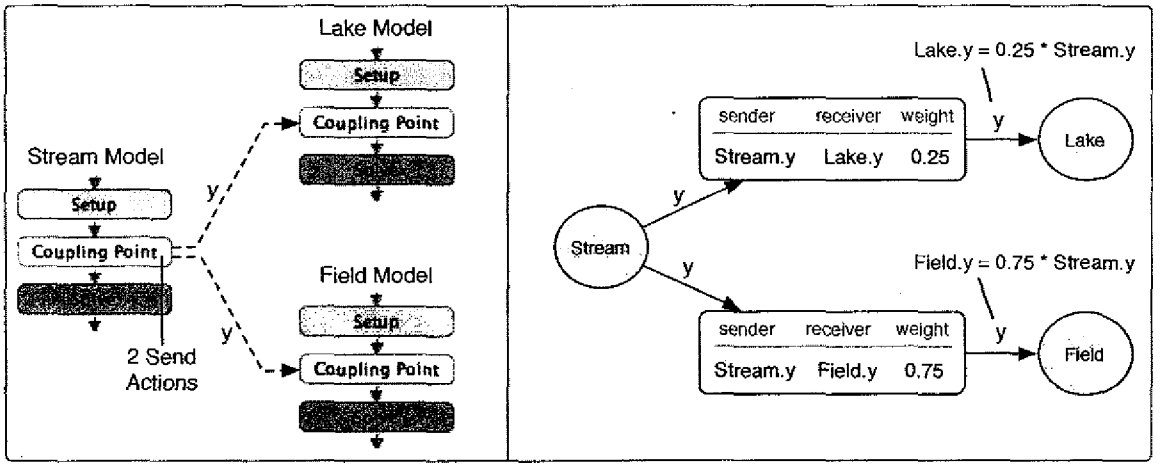

Figure 37. Sending a variable to two different models.

In the figure, the Lake model receives 25% of the value of *y*, and the Field model receives 75% of the value of *y*. In the case where two models each send a variable to another model, two data mappings must again be used, as shown in Figure 38. Both the Stream and River models have a Send Action at their coupling point in the coupling description, each with its own data mapping.

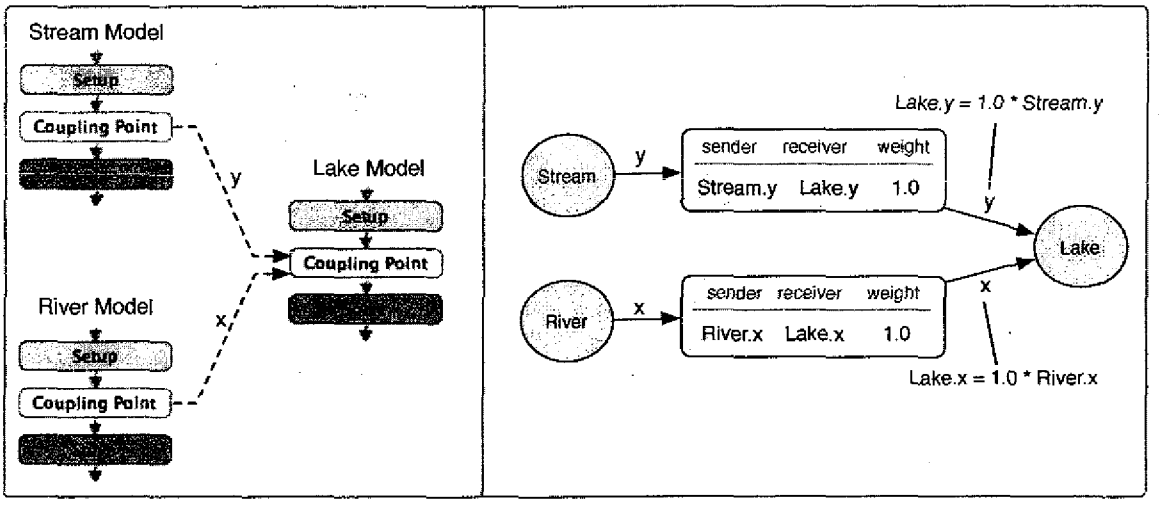

Figure 38. Sending data from two different models.

The data mappings used in the figure indicate that 100% of both the *x* and *y* variables should be sent to the Lake model.

The data mappings in the figure could represent a study site in which a stream and a river both lead into a lake. It is likely though, that there are several streams that lead into the lake. Each of these streams can be simulated by a different instance of the Stream model. This capability of coupling any number of instances of the same model to another model is an important feature of the Coupling Description Language. The PCI serves as the template that describes the behavior of all the instances of a model. The PCI describes how models send to models, while the data mapping describes how instances send to instances. In Figure 39, the coupling description is identical to that of Figure 38, but a different data mapping is used. The upper data mapping in Figure 39 indicates that there are three instances of the Stream model, rather than a single instance as in the upper data mapping of Figure 38.

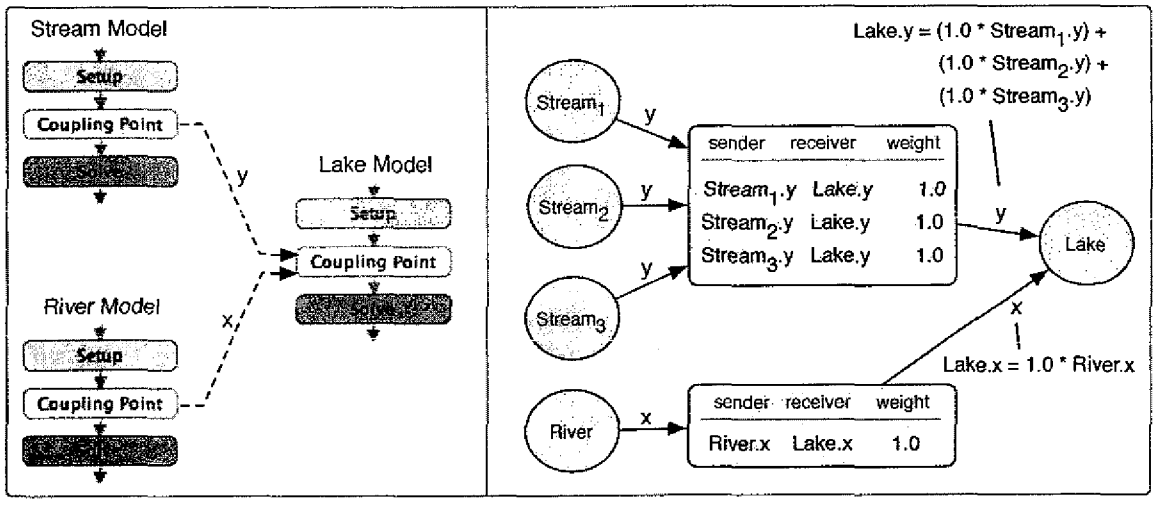

Figure 39. The data mapping indicates there are three instances of the Stream model.

Since the data mapping indicates that there are three instances of the Stream model, then at least three instances of the model will be started when the coupling is executed. In this way, each data mapping is only a partial description of the overall behavior of a coupled model. The global behavior of a coupling is determined collectively by all the data mappings used in a coupling description. The

scientist does not need to explicitly list how many instances there are in a coupling, the total number can be determined from the collection of data mappings.

Since all three instances of the Stream model are sending their value of the same *y* variable, those values must be combined in some way so that only a single value for *y* is received by the Lake model. This is because the Lake model, initially written in a different context, does not expect multiple values. The values of a variable sent from several instances of a model are combined as a weighted sum, as shown in Figure 39 (top-right). Other schemes are possible (such as min, max, average, etc.), but this simple weighted scheme has worked for our domain.

Just as data mappings can be used to specify that there are multiple sending instances, they can also be used to specify that there are multiple receiving instances as shown in Figure 40.

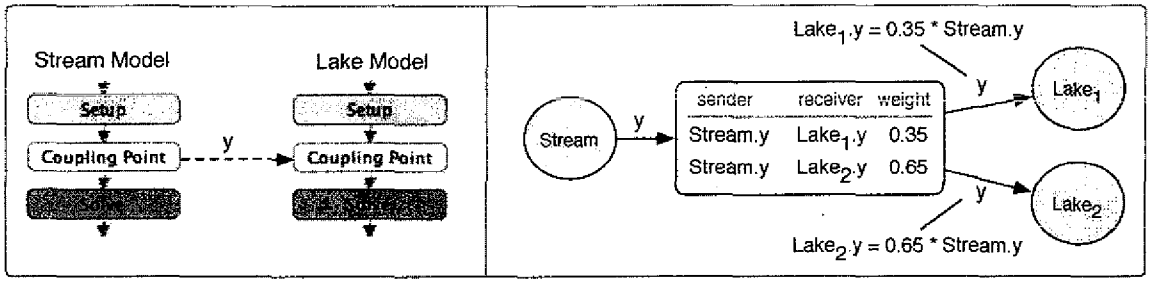

Figure 40. A data mapping that indicates there are two instances of the Lake model.

The data mapping in the figure indicates that there should be two instances of the Lake model, and that 35% of the value of *y* should be sent to the first instance, and that 65% of the value of *y* should be sent to the second instance of the Lake model. The data mapping in Figure 41 indicates that there are two instances of each model.

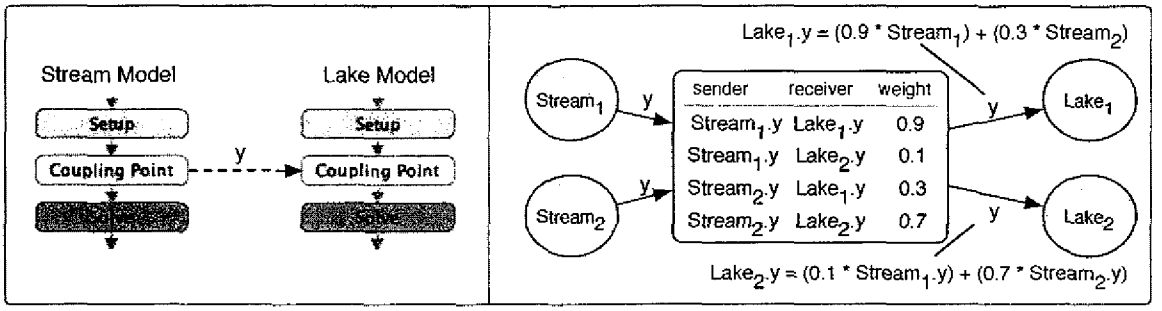

Figure 41. A data mapping that indicates there are two instances of each model.

Instance 1 of the Lake model receives a *y* value that is 90% of the value sent by instance 1 of the Stream model, and 30% of the value of *y* sent from instance 2 of the Stream model. The second instance of the Lake model receives a *y* value that is 10% of the value of *y* of instance 1 of the Stream model, and 70% of the value of *y* of instance 2 of the Stream model.

The data mapping in Figure 41 could represent a study site in which there are two streams, each of which partially leads into two different lakes. It is likely though, that each of these streams is part of a larger network of interconnected streams. Each of the streams in the network can be simulated by a different instance of the Stream model, and those instances can be coupled together. The ability to couple together instances of the same model is an important feature of the Coupling Description Language. Figure 42 shows how three instances of the same model, the Stream model, can send to each other. Each instance in a data mapping can uniquely be identified by a number from 1 to *n*, called the *instance identifier*, where *n* is the total number of instances in a coupled model.

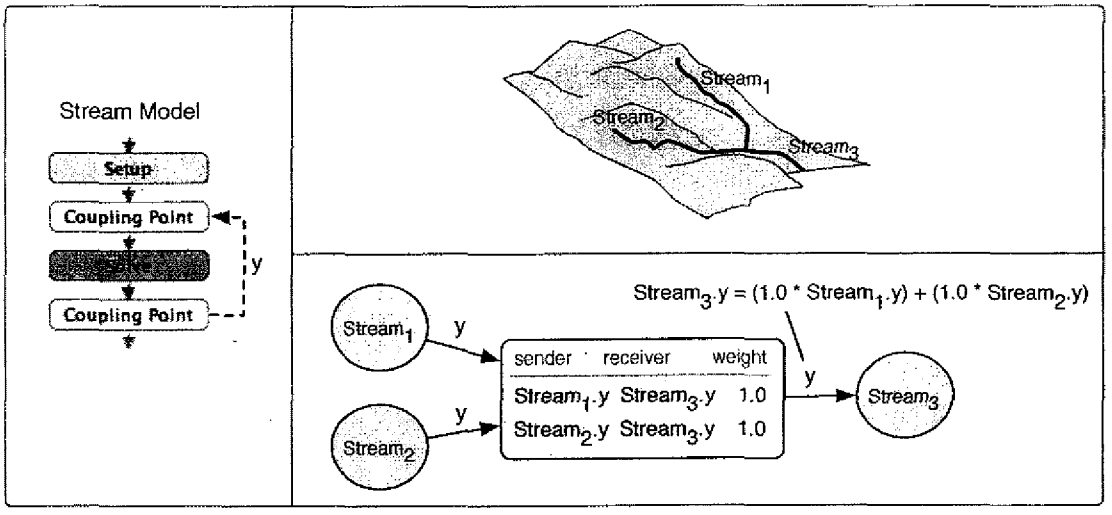

Figure 42. A data mapping that indicates that two instances send to a third instance.

The data mapping in the figure indicates that instances 1 and 2 of the Stream model should send their *y* values to instance 3 of the Stream model. The values from instances 1 and 2 are combined, via a weighted sum, and the final value is received by instance 3. Note that this data mapping indicates that there should be three instances of the Stream model in the coupled model. The topology of which instances send to which is determined directly from the data mapping. A topology involving five instances of the Stream model is shown in Figure 43.

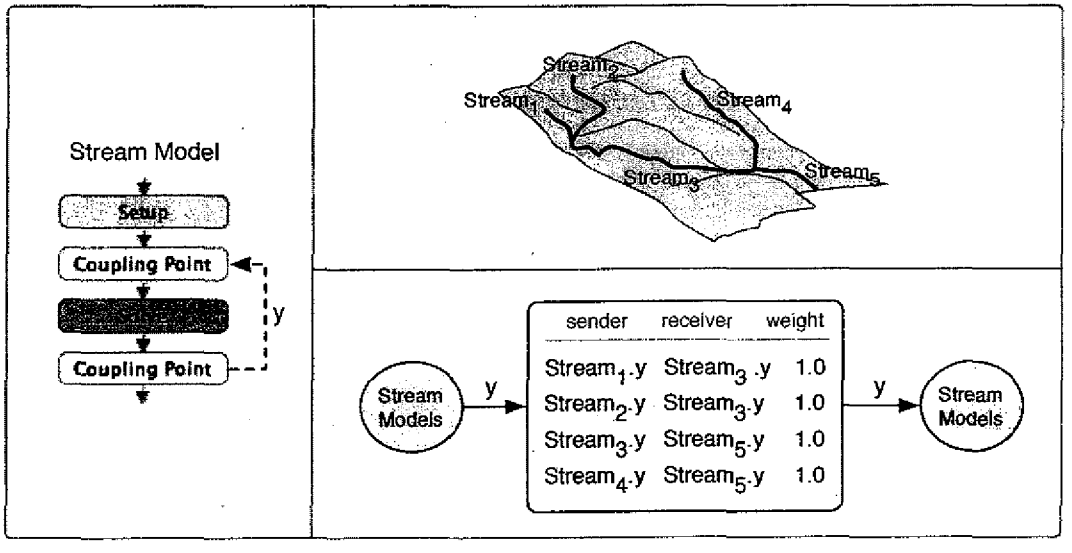

Figure 43. A data mapping that indicates there are five instances of the Stream model.

The figure shows the physical stream network topology, and how it is described in a data mapping. Instances 1 and 2 both send to instance 3, and instances 3 and 4 both send to instance 5. Instance 5 does not send *y*, and instances 1, 2, and 4 do not receive *y*. Notice how only the data mapping was changed in order to change the topology of the instances. Different data mappings refer to the same instances by their globally unique instance identifier. This is shown in Figure 44.

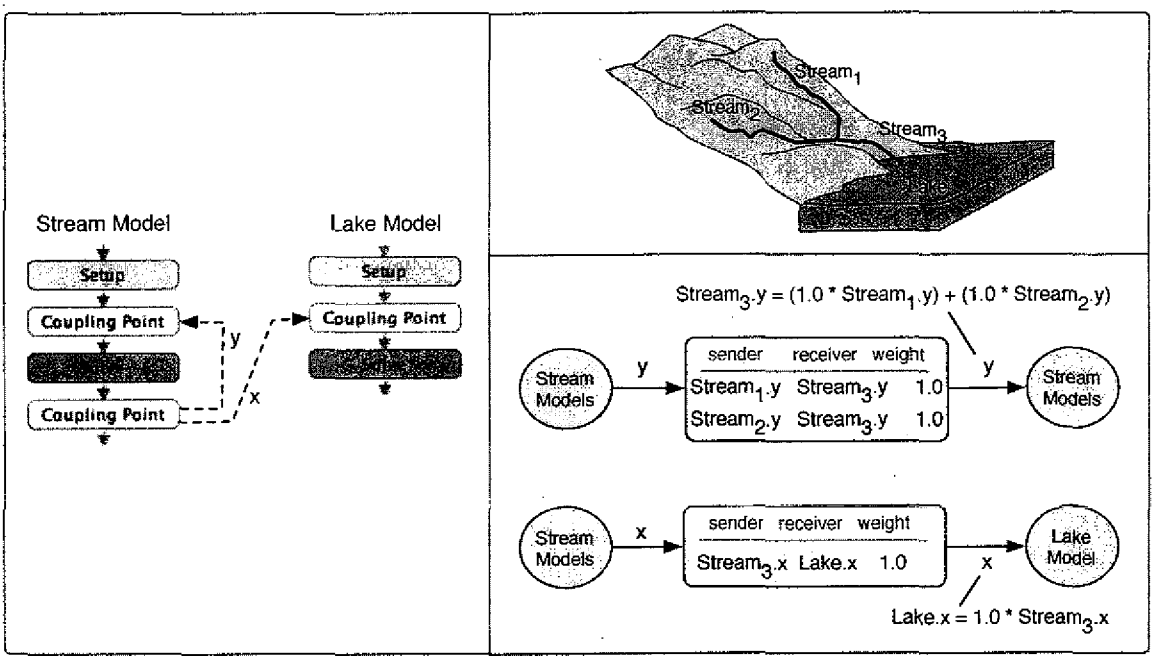

Figure 44. Each instance can be identified by a unique number.

In the figure, there are two Send Actions in the coupling description, each with its own data mapping, shown on the right of the figure. The data mapping of the send of the *y* variable indicates that instances 1 and 2 of the Stream model should both send their value of *y* to instance 3 of the Stream model. The data mapping of the send of the *x* variable indicates that only instance 3 of the Stream model should send its value of *x* to the Lake model. Since the data mapping indicates that the *x* value is sent only from instance 3 of the Stream model, instances 1 and 2 of the Stream model do not perform their Send Actions. Just as some in-

stances may not send a variable, some instances may not receive a variable as well. This is shown in Figure 45.

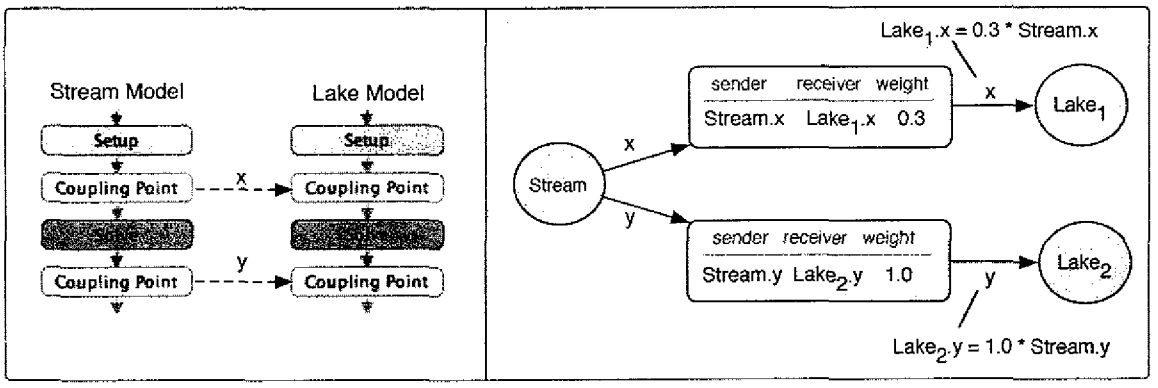

Figure 45. The y variable is sent to only instance 2 of the Lake model.

In the figure, the *x* variable is sent from the Stream model to only instance 1 of the Lake model, and the *y* variable is only sent to instance 2 of the Lake model. Since only instance 1 of the Lake model receives the *x* variable, then only instance 1 is able to perform the actions that use the variable at that coupling point. If *x* is used in an Update Action in the Lake model, then only instance 1 of the Lake model would perform the update function, since only that instance receives a value for *x*. Similarly, since only instance 2 of the Lake model receives the *y* variable, then only that instance performs the actions at that coupling point that use the *y* variable.

The examples in this section showed how data mappings are used to describe the communication between model instances, but they have only considered how a single value is sent. Typically, each Send Action is performed many times during a coupled simulation, resulting in many values being sent. It is important to understand how the models are synchronized through these asynchronous Send Actions. We have adopted a producer-consumer style of communication in which each Send Action results in a single value being produced, which is then consumed by a specific Store or Update Action in another instance. In this

way, there is a strict equality between the number of values produced, and the number of values consumed (each value produced has a specific target action at which it is consumed). This allows the communication to remain synchronized even if there is only one-way communication between the models: the producer model can execute ahead of the consumer sending values, and the order in which they are sent is preserved in the consumer's queue.

For the case of sending array values, data mappings can describe how each individual element of an array is communicated and transformed between instances as well. These data mappings are called *array-level data mappings*, as opposed to *simple data mappings* which have been used in the examples up to this point. Figure 46 shows an example of an array-level mapping.

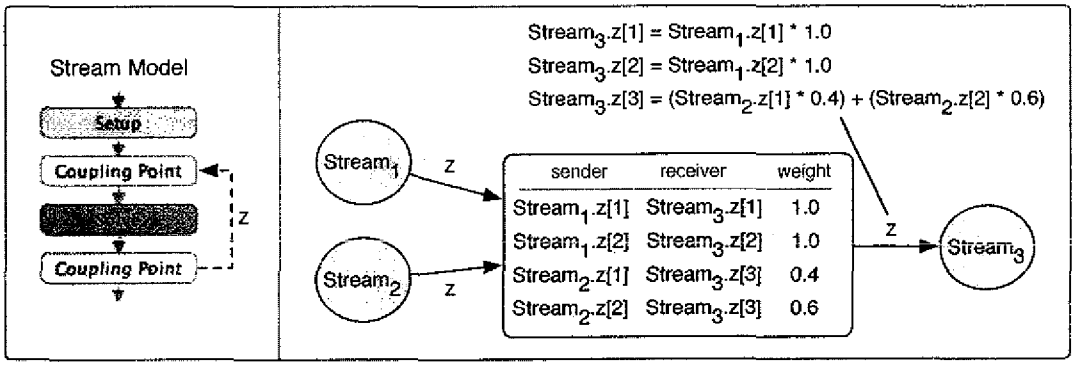

Figure 46. Array-level data mappings allow for a finer grain mapping.

In the figure, the *z* array is sent from instances 1 and 2 of the Stream model to instance 3 of the Stream model. The data mapping indicates that elements 1 and 2 of the *z* variable from instance 1 should be placed into elements 1 and 2 of the array that is received by instance 3. It also indicates that elements 1 and 2 of the *z* variable from instance 2 should be weighted, summed, and placed into element 3 of the array that is received by instance 3. Data mappings are stored in text files and in general are automatically generated (this is demonstrated in Chapter 7), an example of which is shown in Figure 47. A simple mapping is shown at the top of the figure, and an array-level mapping at the bottom of the figure.

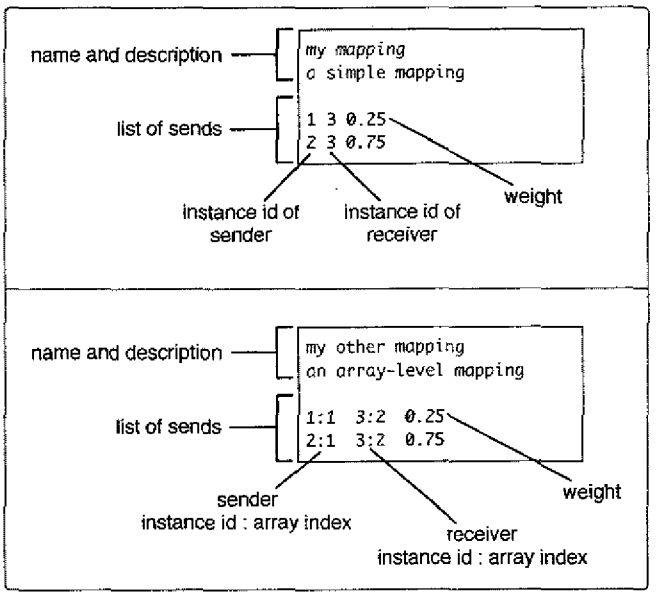

Figure 47. Format of data mapping input files.

This discussion has explained what data mappings are and how they are used, but how are data mappings created? The answer to this question is dependent upon the relationship between the variables of the two models. Consider the example in Figure 48. There are two financial models A and B. Model A simulates the behavior of bank customers, and has the state variable *savings*, an array, the elements of which each represents the predicted amount of money that a different customer possesses in his/her bank account. In this example, element 1 of *savings* represents the amount of money that Joe has in his bank account, and element 2 represents the amount that Kelli has in her account. Model B simulates customer investment behavior and has the state variable *likelihood*, an array, the elements of which each represent the probability that a different customer age range will invest in stocks in the next year. Element 1 of *likelihood* represents the probability that customers in the age range of 18-25 will invest in stocks in the next year, and elements 2 and 3 represent age ranges 26-45 and 45-60 respectively. In this coupling, the user wants to scale the value of the *savings* variable so that it reflects the likelihood that the customers would invest, such that a low

investment likelihood would lower the predicted savings account value for a customer, and vice versa.

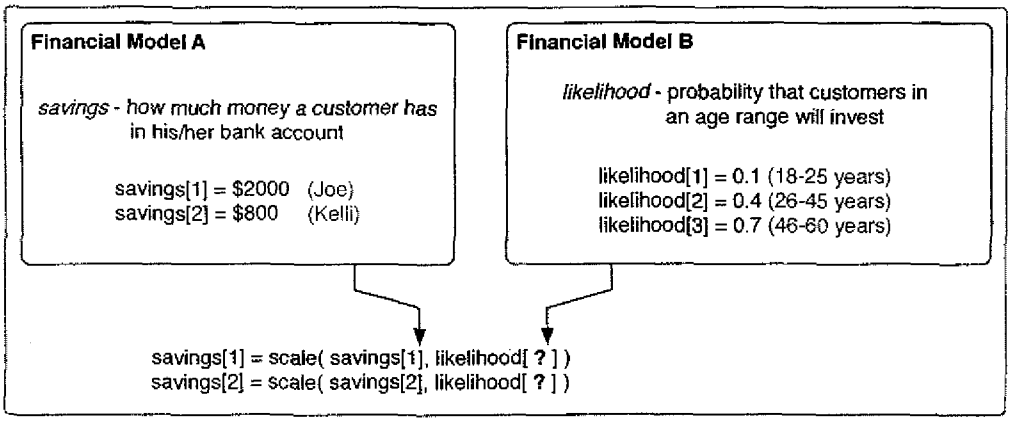

Figure 48. The meaning of variables differs across models.

In order to create the correct data mapping, we need to know the age of each customer represented by the *savings* array. One of the sources of this additional information is databases, but the information could come from any source. Figure 49 shows how a customer database can provide the necessary information to properly relate *likelihood* to *savings*.

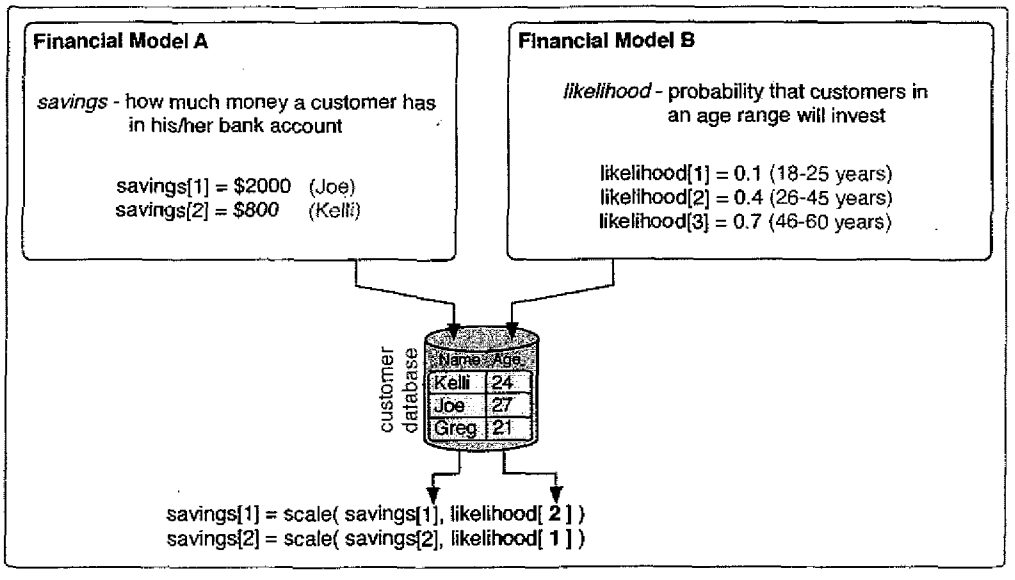

Figure 49. Third-party sources of information can be used to relate variables.

Through the use of the customer database, the ages of the customers represented by the *savings* array can be determined, which can then be used to identify which elements of the *likelihood* array are related. In this example, since Joe is age 27 according to the customer database, then element 2 of the *likelihood* array should be used to update element 1 of the *savings* array. Similarly, since Kelli is age 24, element 1 of the *likelihood* array should be used to update element 2 of the *savings* array.

As another example, consider the models in Figure 50. Model A has an array variable *trans*, the elements of which each represent the volume of water lost via transpiration for a different county. Model B has an array variable *evap*, the elements of which each represent the volume of water lost via evaporation for a different state. In this coupling, the scientist wants to add the amount of water lost via evaporation to each element of the *trans* array.

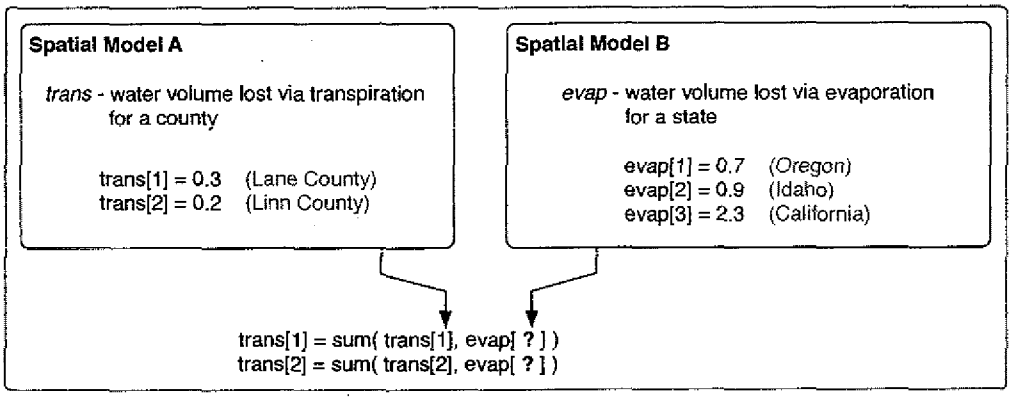

Figure 50. The elements of the two arrays represent different spatial areas.

In order to create the correct data mapping, we need to know how each element of the *evap* array is related to each element of the *trans* array. As in the previous example, a third party can be used to relate the data. Here, a Geographic Information System (GIS) can be used to determine what percentage of each state is covered by each county, a common spatial analysis performed by GIS's known as an *overlay* operation. This is shown in Figure 51.

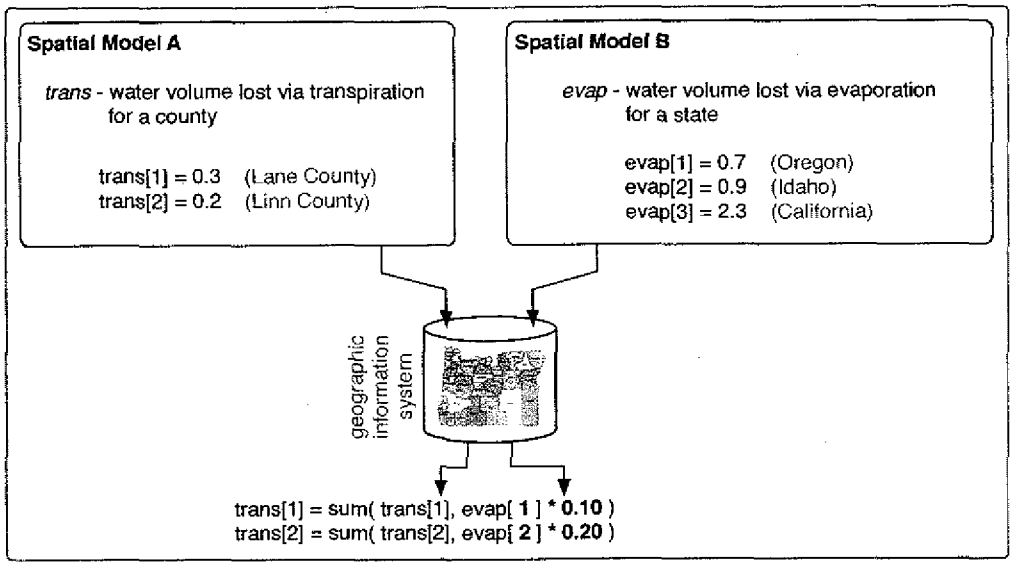

Figure 51. A GIS can be used to relate spatial data.

Through the use of a GIS, the percentage of Oregon made up by Lane County can be determined and then used to properly scale the value of *evap* (to 10% in the figure) so that the proper value of evaporation is added to element 1 of the *trans* array. Similarly, the percentage of Oregon made up by Linn County can also be determined and used to properly scale the value of *evap* (to 20%) that is added to element 2 of *trans*.

In some cases, the creation of a data mapping can be fully automated by taking advantage of the scripting or querying capabilities common in databases. In the spatial data mapping example, a script can be created within the GIS that iterates through each county in the GIS and calculates the percentage of each state that it covers. If the GIS is also told which elements of the *trans* array represent which counties, and which elements of the *evap* array represent which states, then the script can automatically create the complete array-level data mapping in terms of just the array elements. We will see in detail how this is accomplished in the hydrology case studies in Chapter 7.

Models are often written to support the simulation of a range of spatial distributions and scales. This allows the model to be customized for different study sites. A side effect of this is that the physical space represented by the elements of an array can change depending upon the parameterization of a model. This is illustrated in Figure 52. Each element of the array represents a different volumetric cell of a lake.

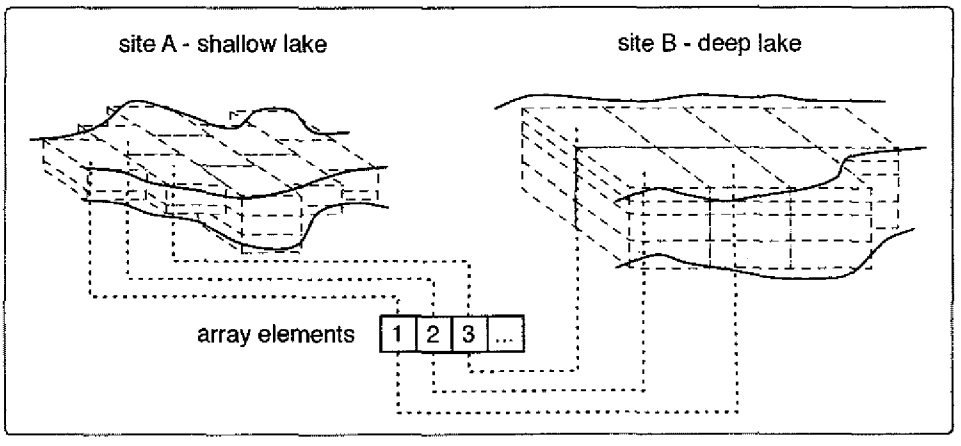

Figure 52. The physical space represented by the elements of an array can change between study sites.

Given one set of model inputs, the array elements each represent a small cell of a shallow lake, but when given a different set of model inputs, the array elements each represent a large cell of a deep lake. Since the physical space represented by the array elements can change from study to study, so too must the data mappings used. Note though that only the data mappings need to be created for each study site, the coupling description itself does not need to change.

This section has explained how data mappings are used to describe the communication between model instances. It was shown how any number of instances of a model can be coupled to another model, and how any number of instances of the same model can be coupled together. It was also shown how data mappings can describe the communication of variable values via simple data mappings, as well as the communication of the individual elements of a

value via array-level data mappings. Next we will see how these exchanges of values can be coordinated in simulation time.

Coordinating Data In Time

In time-dependent models, exchanged data must be related in time as well as meaning. For a model to change its state based on the state of another model, those model states must be representative of the same (or otherwise coordinated) point in simulation time. There are two general approaches to coordinating the models in time: implicitly and explicitly.

Implicit Temporal Coordination

Figure 53 shows two models with typical structures: a setup phase followed by a time step loop. In the implicit approach, the models do not exchange information about their simulation times, or use temporal variables to decide when to communicate. Rather, their loops are coordinated such that they start together at the same point in simulation time, and each iteration of each loop represents the same length of time. The result is that the models will remain coordinated in time, communicating once on each time step. This is shown in Figure 53.

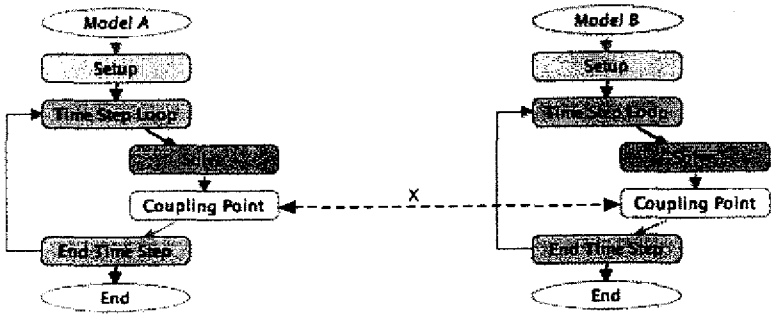

Figure 53. Implicit coordination by matching start times and step lengths.

In the figure, the models communicate at the end of their time step loops. Both start at the same point in simulation time and use the same time step length. If

the models must use different time step lengths, then the greatest frequency at which they can communicate is the least common multiple of the two time step lengths (values of variables at intermediary points in time could be estimated via interpolation/extrapolation). The activation frequencies of the actions must be set accordingly, as shown in Figure 54.

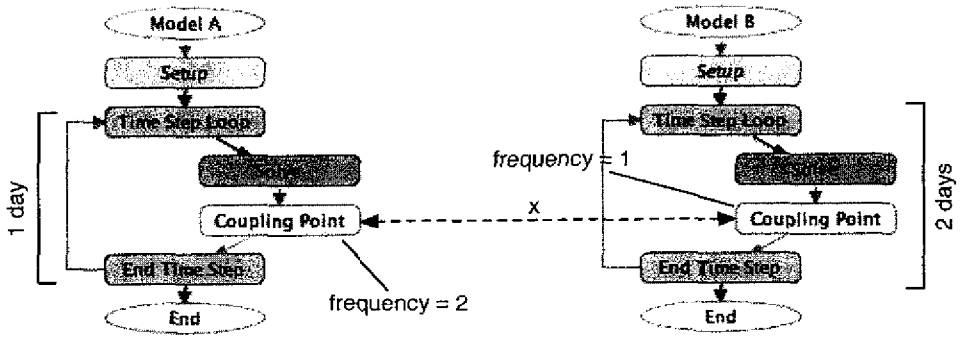

Figure 54. Specifying frequencies to resolve differences in time step length.

In the figure, both models start at the same point in simulation time, and model A uses a time step length of 1 day, while model B uses a time step length of 2 days. In order for the models to remain coordinated, the actions at the coupling point in model A must use an activation frequency of 2, and in model B, an activation frequency of 1. The examples in the next section and the case studies in Chapter 7 use the implicit approach.

The advantage of this approach is that the models do not need to send time information to each other in order to stay coordinated. This is particularly useful because it avoids the need to translate the temporal variables between the models. The way that models represent time varies significantly across models. Some models have variables that keep track of the simulation day, month, year, hour and second, while others simply have one variable that keeps track of the current hour of the simulation. Without implicit synchronization, the temporal variables must be translated. The disadvantage to the implicit approach is that some models use variable length time steps, in which case this approach will not

keep the models synchronized because each iteration of their time loops do not represent the same length of time. In such a case, the models must coordinate their communication times explicitly.

Explicit Temporal Coordination

When one or more models in a coupling is a time-stepped (continuous) simulation with a variable length time step, or is a discrete event simulation (DES), the models must explicitly negotiate the simulation times at which they will communicate. We focus this discussion on DES models, but the coordination issues presented here apply to time-stepped simulations with a variable length time step as well. Figure 55 shows the typical structure of a discrete event simulation.

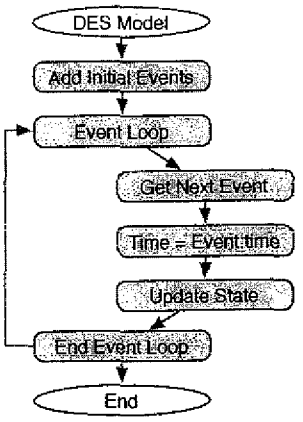

Figure 55. Typical structure of a discrete event simulation.

Typical discrete event simulation models consist of an event queue to which events are added and removed. Each event is associated with a point in simulation time at which the event occurs. An initial set of events is added to the event queue and then the event loop is entered. On each iteration of the event loop, the next event (the one with the minimum time) is removed from the queue and the model's simulation time is advanced to the event's time. The state of the model is then updated in response to the event, which may result in additional events be-

ing added to the event queue. The event loop iterates until there are no more events in the event queue.

In a coupling involving DES models, each constituent DES model would likely communicate with the other models and adjust its state after it handles each event, at the end of the event loop. To illustrate how two simple, typical DES models can be coordinated in simulation time, consider an example in which there are two DES models A and B, that initially have two events in each of their queues which occur at times 12 and 16 in model A, and 7 and 14 in model B. The models are coupled such that Model B sends its state to model A at the end of its event loop, and model A updates its state in response. When the models begin their simulations, each removes the next event, advances its simulation time to the next event's time, and updates its state in response to the event. At the end of the first iteration of the event loop, model A's simulation time is 12, and model B's is 7. Model B then sends its state (which includes its current simulation time) to model A. Since the state of model B represents an earlier point in time than the state of model A, the state of model A cannot be updated based on the current state of model B. Rather, model A must save the state that it received and wait until model B processes more events and advances its time to 12 (or some later point). Model B would handle its next event, which occurs at time 14, and send its state to model A. Since model B now represents a later point in time than model A, model A can update its state based on the saved state of model B that represents the state of model B at time 7. This is acceptable because the state of model B is the same at time 7 as it is as time 12 (since no events occur in model B until time 14). Model A then stores the new state of model B and the interaction repeats.

This simplified example demonstrates how the current simulation time of a DES model cannot be anticipated by other coupled models, requiring the models to exchange temporal information. In these cases, the activation frequencies of

actions are set to 1 so that the models communicate every time a coupling point is reached. The update functions interpret the temporal state of the other models and behave accordingly.

Examples

To illustrate how the CDL can be used to describe coupled models, two examples follow. Chapter 1 presented two ways in which comprehensive models can be created by coupling existing models. The first way is to incorporate the simulation of additional physical processes by coupling different models. The first example demonstrates this. The second way is to couple together many instances of a single model, accounting for the interactions between instances. The second example demonstrates this. In both examples, we follow the six steps to creating a coupled model described in Chapter 3 and we focus on how the process is carried out in our approach. Since we have already chosen the models (Step 1), the process will begin at Step 2 in each example. In the first example we begin with a discussion of the basic hydrological concepts upon which the examples and case studies (in Chapter 7) are based.

Example One

A common use of modeling in the field of hydrology is in the study of how rainfall affects surfacewater, such as rivers and streams. One such rainfall-runoff model is the Storm-water Management Model (SWMM) (Huber and Dickinson 1988), which simulates the amount of surfacewater runoff that exits an area of land in response to rainfall. It was developed by the U.S. Environmental Protection Agency and consists of approximately 30,000 lines of Fortran source code. The physical system simulated by SWMM is illustrated in Figure 56.

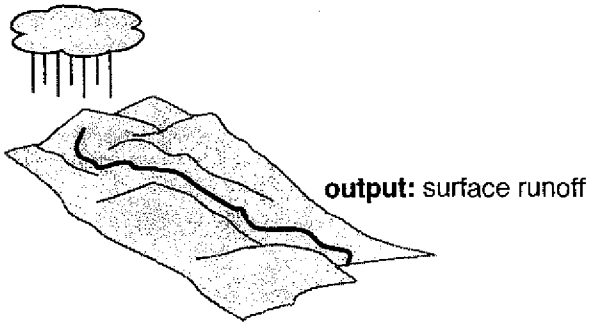

Figure 56. Illustration of the physical system simulated by SWMM.

The amount of runoff that is generated in response to a storm event is not only dependent upon the intensity of the rainfall, but it is also dependent upon the characteristics of the land upon which it falls. The slope of the land and the type of ground cover (grass, forest, urban, etc.) are salient characteristics that are taken in to account in the model. Another important characteristic is the depth of the groundwater beneath the land, called the *water table* or *head*. If the water table is close to the surface, then water is drawn upward and contributes to the amount of runoff, decreasing the amount of groundwater. This upward movement of water is called *baseflow*. Conversely, if the groundwater is deep below the surface, then runoff is drawn downward and contributes to the quantity of groundwater, decreasing the amount of runoff. This downward movement of water is called *recharge*. This water flux occurs through the *unsaturated zone*, located just beneath the land surface. In the unsaturated zone, the very small spaces between particles of dirt and sand are filled partially by air and partially by water. The upper part of the unsaturated zone is called the *root zone*. Below the unsaturated zone is the *saturated zone*, in which the small spaces between particles are filled entirely by (ground)water. The upper limit of the saturated zone is the water table. An illustration of these zones and the water flux between them in shown in Figure 57.

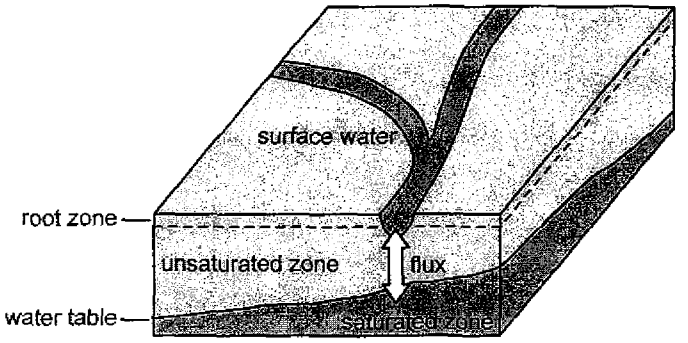

Figure 57. Water flux occurs through the unsaturated zone.

There are three possible relationships between the water table, root zone, and land surface, as shown in Figure 58.

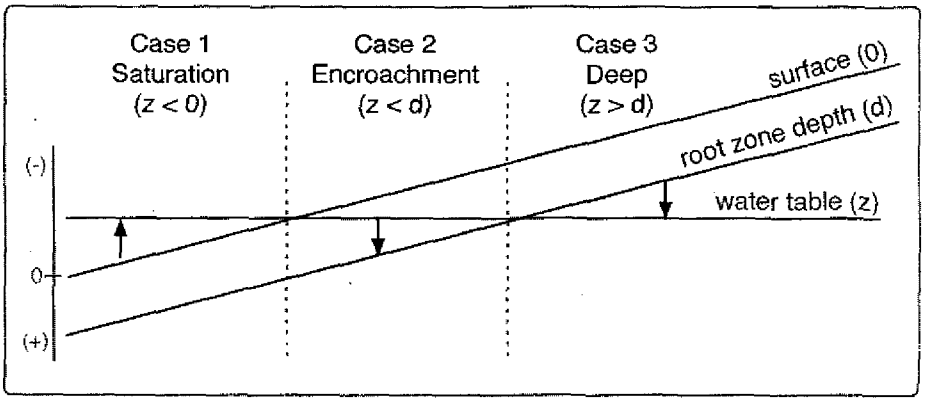

Figure 58. Relationships between the water table, root zone, and the surface.

The water table can either be below, within, or above the unsaturated zone. Each case is described.

Case 1: the water table is above the surface, $z < 0$, resulting in the surface being saturated. The excess water in these areas becomes (saturation-excess) runoff, and the root zone moisture content is saturated.

Case 2: the water table encroaches into the root zone, where $0 < z < d$. In these areas, the soil moisture content is increased due to the presence of the shallow groundwater. The increase in soil moisture results in both a reduced infiltration capacity, which may result in increased (infiltration-

excess) runoff, and an increase in moisture available for plants, possibly increasing evapotranspiration.

Case 3: the water table is deep, where $z < d$, and the groundwater does not affect the root zone. There may be some recharge to the groundwater from the root zone, depending upon the amount of water present.

In hydrological modeling, it is imperative that the interaction between surfacewater and groundwater is well understood. In many situations, it is essential for simulations to fully account for the impact of surfacewater dynamics on groundwater dynamics and vice-versa. Most hydrological models either accurately simulate surfacewater dynamics and poorly simulate groundwater movement, or accurately simulate groundwater movement and do not fully account for surfacewater dynamics. SWMM is a surfacewater model that does not accurately simulate groundwater movement. SWMM calculates changes in the water table elevation using simplified flow calculations. A more accurate simulation can be achieved if a more accurate water table elevation is simulated.

ModFlow (McDonald and Harbaugh 1999) is a widely used hydrological model that simulates the movement of groundwater where the primary output is the water table head of the groundwater. It was developed by the U.S. Geological Society and consists of approximately 10,000 lines of Fortran source code. Figure 59 illustrates the physical system simulated by ModFlow.

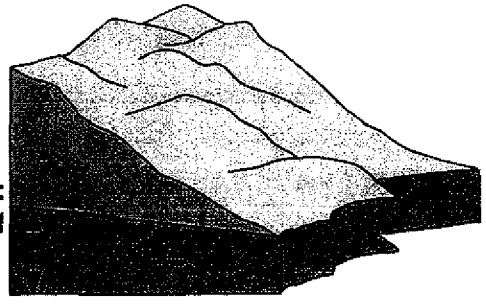

Figure 59. Illustration of the physical system simulated by ModFlow.

In this example, we show how SWMM can be coupled to ModFlow such that the water table head simulated by ModFlow is used by SWMM in place of SWMM's own simplified groundwater calculations. This is achieved in the coupled model by overwriting the water table value calculated by SWMM with the water table value simulated by ModFlow. This example is based on an existing coupling between these models performed by Rowan (2001), although to simplify our discussion we show only how ModFlow affects SWMM but not the reverse.

The Participating Variables

The first step in creating the coupled model is to identify which state variables in the ModFlow and SWMM PCIs represent water table elevation. Inspection of the model PCIs reveals that this variable is called *stg* in SWMM, and *hnew* in ModFlow. As described in the SWMM PCI, *stg* represents the water table head beneath an irregularly-shaped 2d area of land called a *subcatchment*. A subcatchment is an area of land that drains to a single point. Figure 60 shows how subcatchments appear in the ArcMap GIS.

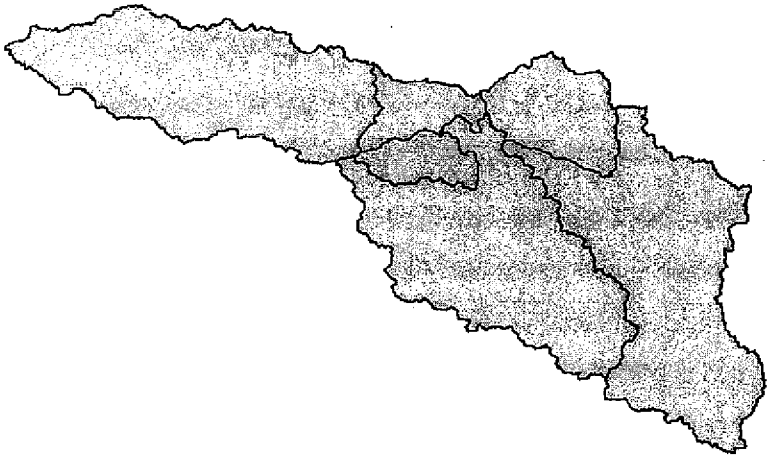

Figure 60. Subcatchments are irregularly shaped polygons.

The *stg* variable is an array in which each element represents the water table head beneath a different subcatchment. As described in the ModFlow PCI, *hnew* represents the water table head of a body of groundwater called an *aquifer*. An

aquifer is a 3d volume of groundwater with defined lateral and vertical extents. The *hnew* variable is an array in which each element represents the water table height of a different regularly-shaped 3d cell. The units of both variables are in feet, as referenced to some known benchmark such as mean sea level. Figure 61 shows a single subcatchment superimposed on a single rectangular grid cell, as it appears in the ArcMap GIS.

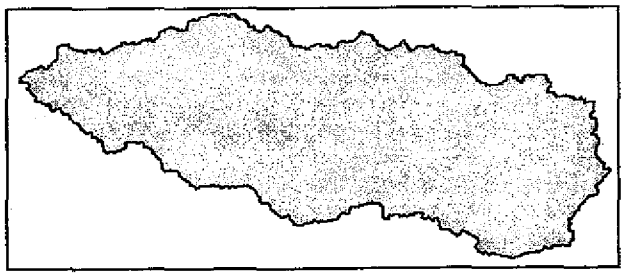

Figure 61. A single subcatchment superimposed on a single grid cell.

This difference in spatial distribution of the participating variables is common in couplings and must be taken into consideration in the coupled model design. In the CDL, such incompatibilities are resolved through data mappings. Before we explain how these spatial differences are resolved, we first present the coupling description. We first show how the models can be coupled where SWMM is parameterized to simulate a single subcatchment and ModFlow is parameterized to simulate a single grid cell. We then show how the coupling can be modified to support the simulation of several grid cells by ModFlow.

Creating the Coupling Description

The coupling description is shown in Figure 62. As in most couplings, the models communicate within their time step loops.

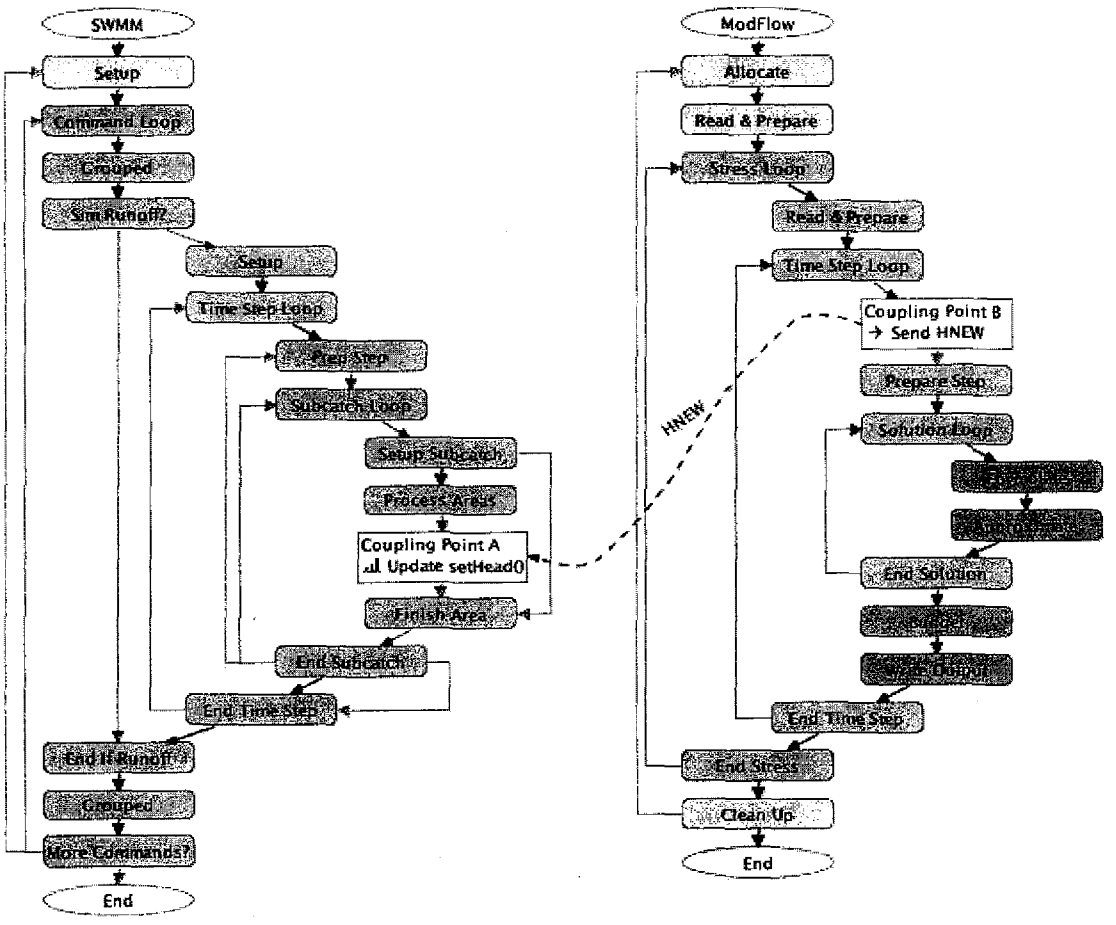

Figure 62. The coupling description as shown in PCICouple.

As long as the models start at the same point in simulation time, and use the same time step length, then the models will remain coordinated in simulation time. ModFlow though, usually uses a long time step length, on the order of days or weeks due to the slow speed at which groundwater moves, and SWMM can use either a short step length to study individual storm events (on the order of hours), or a longer step length to study long term trends (on the order of days). Differing step lengths can be accommodated by adjusting the activation frequencies of actions. For example, if ModFlow uses a step length of 2 days, and SWMM uses a step length of 1 day, then the greatest frequency at which the models can communicate is every 2 days, and the activation frequencies of the

actions in SWMM would be set to 2, and set to 1 in ModFlow. To keep this example simple, we parameterized the models to use a common step length of 1 day.

At the start of the time step loop in ModFlow, the value of the *hnew* variable represents the current water table head, so we expanded the coupling point located there and labeled it "Coupling Point B". The value of *stg* is calculated and set within the "Process Areas" block in the SWMM PCI, so it is immediately after this block that the value must be overwritten with the value of the *hnew* variable received from ModFlow. We expanded the coupling point that follows this block and labeled it "Coupling Point A". We then added a Send Action from Coupling Point B to Coupling Point A that sends the value of the *hnew* variable from ModFlow to SWMM. We added an Update Action at Coupling Point A that applies a custom update function called *setHead*, written in Fortran (added to the coupling description following the process explained in Appendix D), that assigns the value of the *stg* variable to the value of the *hnew* variable received from ModFlow. The function assumes that the units of the variables are the same and it accounts for the difference between the reference datum from which two variables are measured by adding 5.0 feet to *hnew*. The source code for the update function is shown in Figure 63.

```
subroutine setHead(instanceID,dest,src)
  integer instanceID
  real    dest,src

  dest = src + 5.0
end
```

Figure 63. The source code for the setHead update function.

Notice though, that Coupling Point A is located within the subcatchment loop in SWMM. Since SWMM is parameterized to simulate only a single subcatchment, the body of this loop is executed only once on each time step. If, however, SWMM is parameterized to simulate more than a single subcatchment,

Coupling Point A would be reached several times on each time step, once for each subcatchment, and hence SWMM would expect ModFlow to send the value of the *hnew* variable several times on each time step. Since ModFlow only sends this value once on each time step, the models would become unsynchronized during execution of the coupled model. To resolve this structural incompatibility, the activation frequency of the Update Action must be set equal to the number of subcatchments. In this way, activation frequencies can be adjusted to accommodate both differing time step lengths and differing loop structures between models.

The details of the Send and Update actions are shown in Figure 64 as they appear in PCICouple's inspector window.

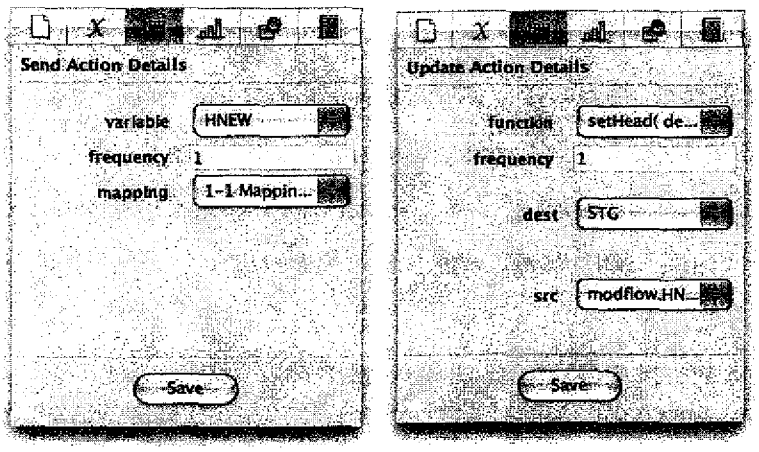

Figure 64. The details of the Send and Update Actions.

The Send Action uses the default data mapping provided by PCICouple. This data mapping is shown in Figure 65.

```
1-1 Mapping (default)

1 2 1.0
```

Figure 65. The default 1-to-1 data mapping.

The mapping indicates that there is a single instance of each model, and that instance 1 sends the value of the variable to instance 2 with a weight of 1.0 (the value is unchanged).

Thus far we have only considered the case where ModFlow is parameterized to simulate a single grid cell. We now explain how this coupling can be modified to account for the case where ModFlow is parameterized to simulate a grid of cells.

Incorporating the Spatial Distribution of Physical Quantities

ModFlow is capable of spatially distributed simulation in which the water table head is simulated for many cells of a grid. Figure 66 is similar to Figure 61, but shows how the area beneath the subcatchment can be divided into 4 cells.

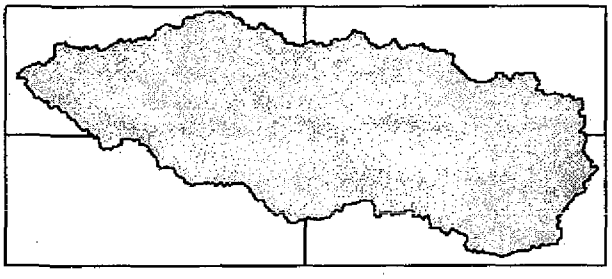

Figure 66. The area beneath the subcatchment is discretized as four cells.

In this part of the example we parameterize ModFlow to simulate four grid cells. As a result, the *hnew* array contains four elements, each of which represents a different grid cell. The *setHead* update function though, only accepts a single scalar value as the input from ModFlow. For this reason, we must change the data mapping used so that the four values of ModFlow's *hnew* array are combined to form a single water table value representative of the area below the subcatchment, which is then received by SWMM and used in the update function. Note that we could also modify the *setHead* function to accommodate receiving an array from ModFlow, but such customizations to update functions for specific couplings is not desirable since it limits the reusability of the functions.

The new data mapping is shown in Figure 67. Unlike the previous mapping, this is an array-level mapping which indicates how the individual elements of the variable are transformed and communicated.

```
4 Cell-to-1 Subcatchment Mapping

1:1 2:1 0.25
1:2 2:1 0.25
1:3 2:1 0.25
1:4 2:1 0.25
```

Figure 67. The array-level data mapping.

The mapping indicates that the four elements of the variable being sent are each scaled to 25% and then summed into a single scalar value before being received by instance 2. If SWMM is parameterized to simulate multiple subcatchments, the data mapping can be be modified accordingly. Figure 68 illustrates how the elements of the array are combined and sent from ModFlow to SWMM.

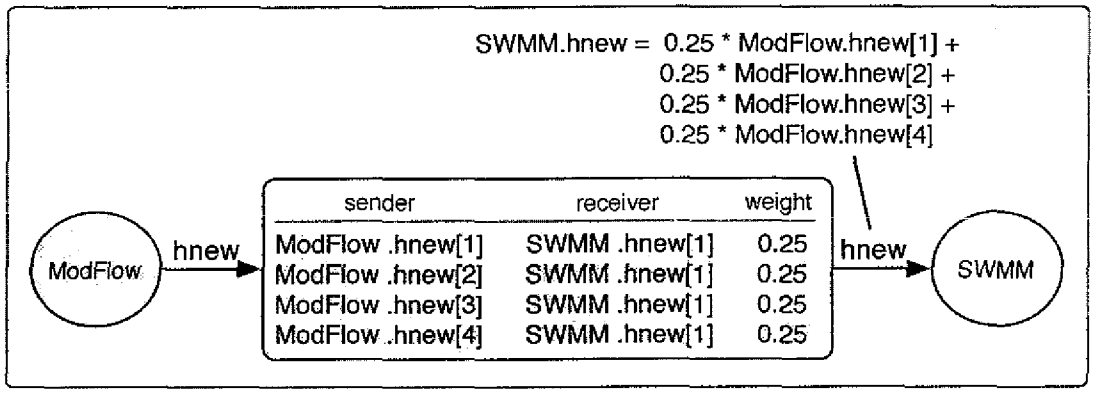

Figure 68. How the array elements are combined and sent.

The CDL is capable of describing couplings between any number of models. We explain next how a third model can be added to this coupling.

Incorporating Another Model

Suppose that snow accumulates over the lower-right area of our hypothetical study site. As the snow melts, it infiltrates into the ground and recharges

the groundwater below. We can account for this in the coupled model by coupling a snowmelt model to ModFlow.

The Utah Energy Balance (UEB) (Tarboton and Luce 1996) snowmelt model simulates the amount of water produced as a result of snowmelt for the given meteorological conditions. It is written in Fortran and consists of approximately 1,000 lines of source code. The physical system simulated by UEB is illustrated in Figure 69.

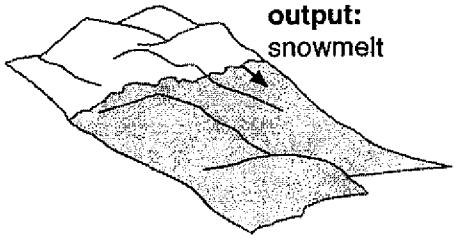

Figure 69. Illustration of the physical system simulated by UEB.

UEB is a lumped model, so it calculates a single snowmelt value that is representative of the snowmelt over the entire area being simulated. Temperature variation throughout the day plays an important role in the simulation of snowmelt, so the length of time step used in this model is typically one hour. The updated coupling description is shown in Figure 70.

We parameterized UEB to simulate the area of land located above the lower-right cell of ModFlow, and we added the PCI for UEB to our coupling description. According to the PCI for UEB, the *q* variable, a scalar, represents the amount of snowmelt (as a height in feet) so it is the value of this variable that must be sent to ModFlow. It should be sent at the end of the time step loop in UEB, so we expanded the last coupling point in the loop and labeled it "Coupling Point C". We added a Send Action from this coupling point to Coupling Point B that sends the value of the *q* variable to ModFlow.

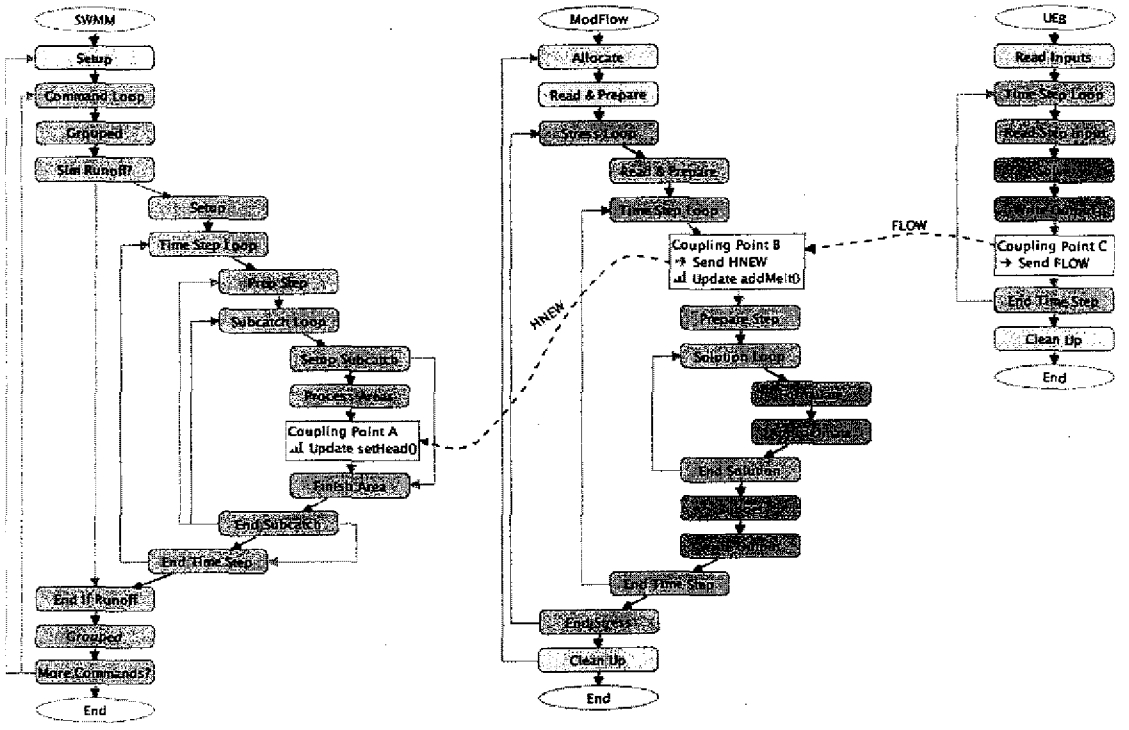

Figure 70. The updated coupling description.

We added an Update Action at Coupling Point B that applies a custom update function called *addMelt* that adjusts the water table head, *hnew*, based on the snowmelt value received from UEB. The source code for the *addMelt* function is shown in Figure 71.

```
subroutine addMelt(instanceID,head,melt)
  integer instanceID
  real    head(2,2),melt

  head(2,2) = head(2,2) + melt
end
```

Figure 71. The source code of the addMelt function.

The function adds the melt value to element 2,2 of *hnew*, which is the element that represents the grid cell located beneath the area simulated by UEB. Notice that the after the Update Action was added to Coupling Point B, it was reordered

to occur before the existing Send Action at that point. This way, the effect of UEB on ModFlow indirectly affects SWMM.

The details of the Send and Update Actions are shown in Figure 72. Notice that the Send Action has an activation frequency of 24 since UEB uses a time step length of 1 hour, yet it should only communicate with ModFlow once per day.

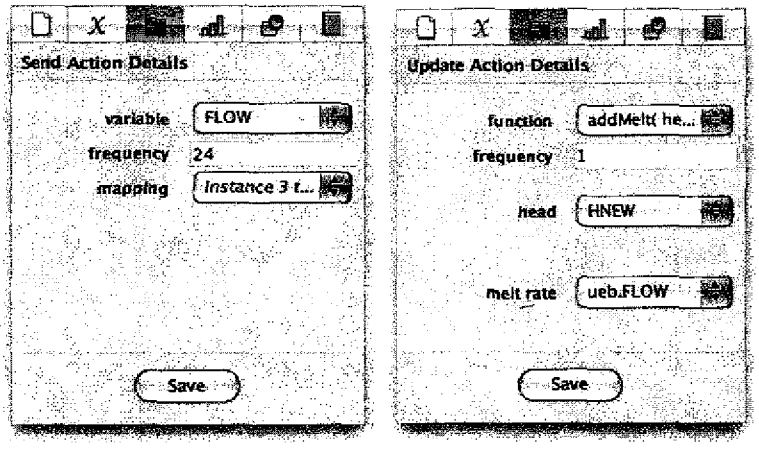

Figure 72. The details of the Send and Update Actions.

This example has demonstrated how models can be coupled together to create more comprehensive simulations of physical systems. We saw how ModFlow and SWMM could be coupled first in a simple way, where each model was parameterized as a lumped model, and then we saw how the spatial-distribution simulation capabilities can be supported in couplings. We then saw how an additional model could be added to the existing coupling, incorporating the simulation of additional physical processes into the coupled model. The original coupling between SWMM and ModFlow can be thought of as a model itself, to which UEB was coupled. In the next example, we show how many instances of the same model can be coupled together to achieve spatially distributed simulation.

Example Two

In this example we show how a lumped-parameter model can be coupled to itself to achieve a spatially distributed simulation. We consider a hypothetical study site in which there is an agricultural field that is subject to a variety of management practices including fertilization and irrigation. The fertilizer introduced into field is transported by the movement of water into the unsaturated zone and along the surface as runoff. The Groundwater Loading Effects of Agricultural Management Systems (GLEAMS) (Leonard, Knisel, and Still 1987) model provides the simulation of the movement of water, sediment, pesticides, and nutrients based on meteorological and management practice inputs. This system is illustrated in Figure 73.

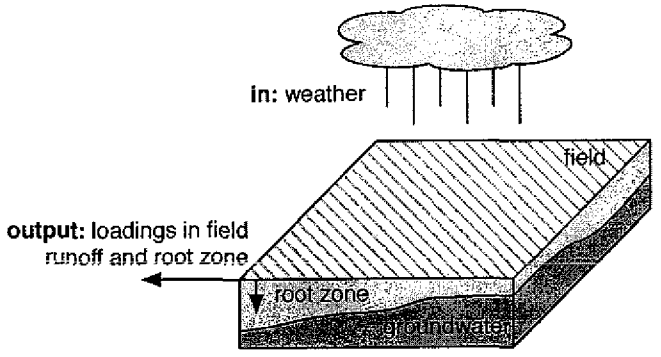

Figure 73. Illustration of the physical system simulated by GLEAMS.

The time step length used by the model is fixed at one day in length, and the length of time simulated is typically multiple years to account for seasonal variation in the model inputs and to assess long term trends.

Since GLEAMS is a lumped-parameter model, all model inputs and outputs are single values representative of the entire field. If the study field has homogeneous characteristics, then the model is appropriate for simulation of the field. If however, some characteristics of the field vary spatially, such as the concentration of a particular nutrient, then the study field must be divided into homogeneous parts, and each part must be simulated individually using individual in-

put parameter sets for each model run. If, for example, the nutrient concentration varies significantly across two parts of the field, as in Figure 74, then two separate simulations would have to be performed, one on the upper half of the field and one on the lower half.

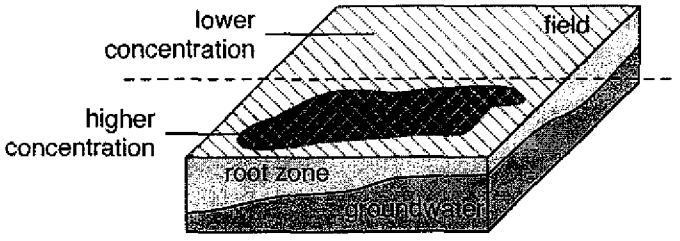

Figure 74. Lumped models do not support heterogeneity.

Independent simulations of the field would result in separate simulations of the groundwater, possibly resulting in significantly different water table heads, even though the groundwater beneath the entire field is connected and continuous. To avoid this discrepancy, we couple several instances of GLEAMS together so that each instance simulates a different, homogeneous, part of the field, while at the same time maintaining a consistent water table head. We begin by showing how to couple two instances of the model, one of which simulates the lower half of the field, and the other which simulates the upper half.

The Participating Variables

Since the interested physical quantity is water table head, the variable that represents this physical quantity must be exchanged between the instances of the model. By inspection of the PCI for GLEAMS, the soil water content variable, *st* is identified. The variable is an array in which each element of the array represents the soil water content at a different soil layer (depth in feet), over the full area of the field. The vertical soil layers are illustrated in Figure 75. Now that the participating variable has been identified and understood, the next step is to determine the location within the model code where this variable should be accessed.

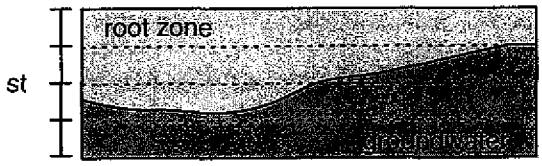

Figure 75. The spatial distribution of the variable.

Creating the Coupling Description

The coupling description is shown in Figure 76. The PCI for GLEAMS shows that there are two temporal loops, an outer loop that iterates over years, and an inner loop that iterates over days. As in the previous example, the instances communicate within their time step loops. We expanded the coupling point at the end of the daily loop and labeled it "Coupling Point A". We then added a Send Action and an Update Action to this point. The *st* variable is initialized in the "Setup" block and then updated throughout the "Solve" block on each daily time step.

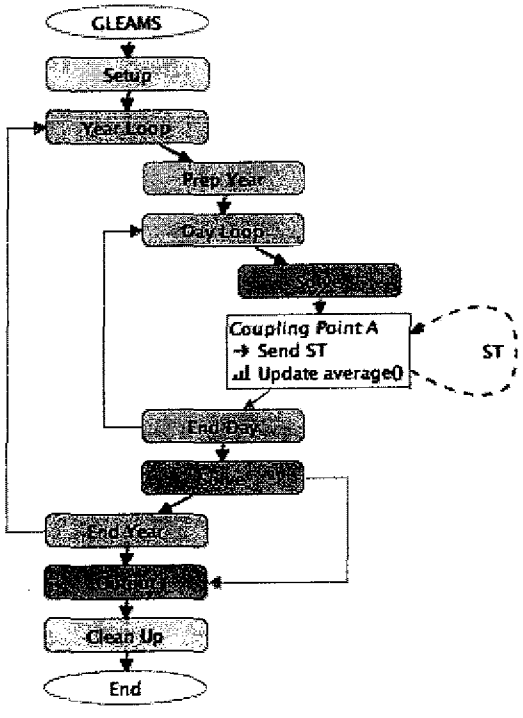

Figure 76. The coupling description as it appears in PCICouple.

We want instance 1 to send to instance 2, and vice versa, so we wrote a simple data mapping by hand that indicates there are two instances, and that each one sends to the other. This mapping is shown in Figure 77.

```
GLEAMS Mapping - 2 Instances
The simple data mapping for example 2
1   2   1.0
2   1   1.0
```

Figure 77. The data mapping used by the Send Action.

When each instance receives the value of the *st* variable that was sent from the other instance, it must average that value with its own value of *st*. To accomplish this, we added an Update Action that applies the built-in *average* update function to these values. The details of the Send and Update Actions are shown in Figure 78 as they appear in PCICouple.

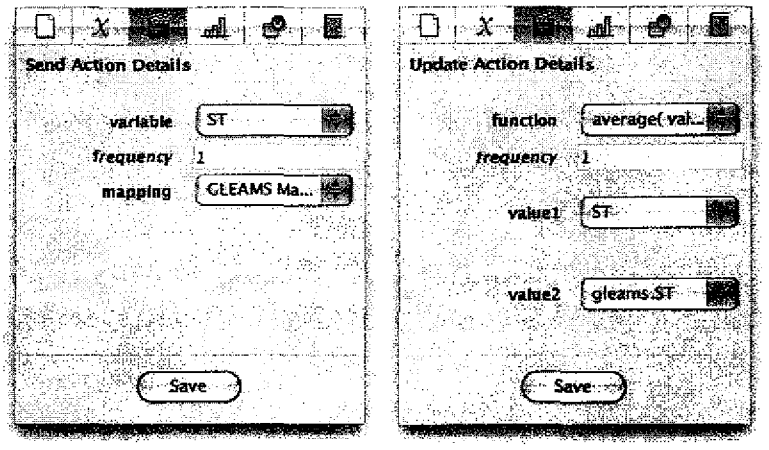

Figure 78. The details of the Send and Update Actions.

The Send and Update Actions were deliberately ordered such that the Send Action occurs prior to the Update Action. When each instance reaches the coupling point, the Send Action is activated and the value of the *st* variable in each instance is sent to the other instance immediately (recall that data is sent asynchronously). Each instance then waits to receive the value from the other instance before carrying out the Update Action. If however, the Update Action

were placed before the Send Action, each instance would stop and wait at the Update Action and never reach the Send Action, deadlocking the coupled model.

If executed as is, both instances will perform the same exact simulation because they will both read the same model input file. What we want though, is for each instance to use a different input file that describes the half of the study field that it simulates. To accomplish this, we added an Update Action at the "Setup" block that applies the custom update function *makeUnique* to the variable that holds the filename of the nutrient input file, *nutin*. The built-in *makeUnique* function prepends the instance identifier to the start of a filename, making it unique. The revised coupling description is shown in Figure 79.

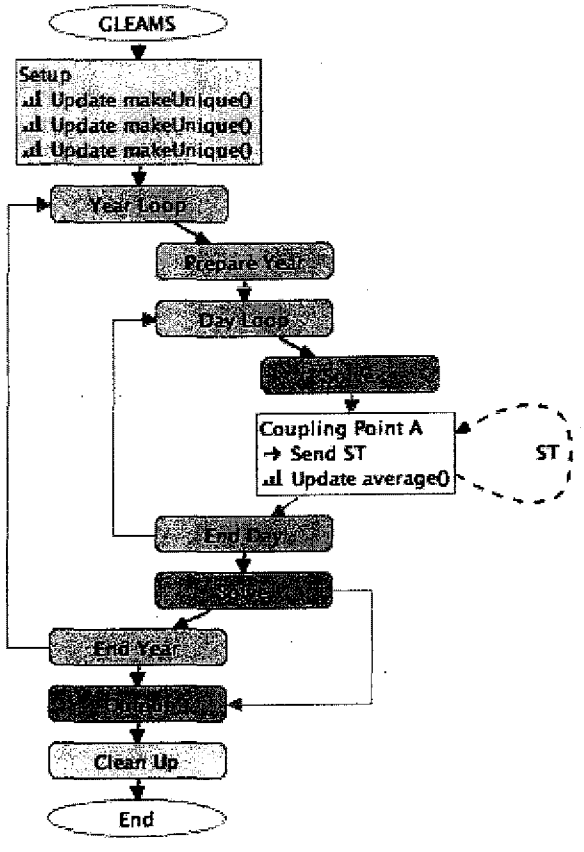

Figure 79. The final coupling description.

We added two additional Update Actions at the "Setup" block that apply the *makeUnique* function to each output filename as well, *hydout* and *eroout*; otherwise, both instances will attempt to write its output to the same file, resulting in runtime errors. The details of the Update Action that operates on the *nutin* variable is shown in Figure 80.

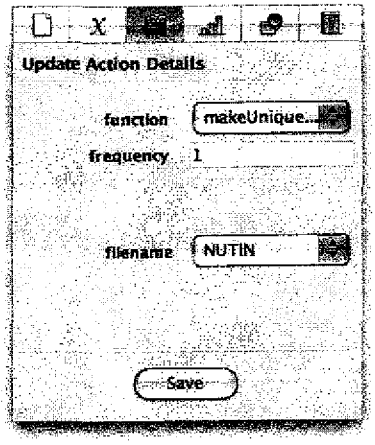

Figure 80. Details of the Update Action applied to the nutin variable.

When the coupled model is executed, two instances of the GLEAMS model are started and at the end of each simulation day, the instances exchange their values of the *st* variable, average them, and maintain a consistent water table head throughout their simulations. Thus far we have only considered the case where there are two instances of the model. In the next section we show how the coupling description can be easily extended to support the coupling of several instances.

Simulating More Areas

In the previous section we showed how a field can be divided into two parts, each of which is simulated by a different instance of the model. It is likely though, that the field has several heterogeneous areas, each of which must be simulated by a different instance as shown in Figure 81.

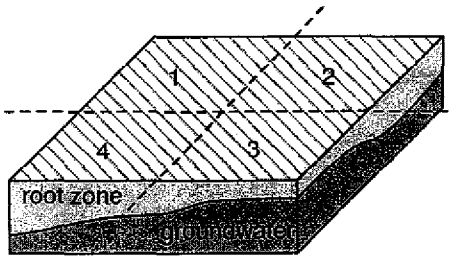

Figure 81. A field with four parts.

To accommodate the involvement of four instances, the coupling description must only be changed such that the Send Action uses a different data mapping. This mapping is shown in Figure 82.

```
GLEAMS Mapping - 4 Instances
Four instances, arranged as a 2x2 grid
1   2   0.5
1   4   0.5
2   1   0.5
2   3   0.5
3   2   0.5
3   4   0.5
4   1   0.5
4   3   0.5
```

Figure 82. An alternative data mapping indicating four instances.

We wrote this simple mapping by hand which indicates that there are four instances and that each instance communicates with its neighbors. This is illustrated in Figure 83.

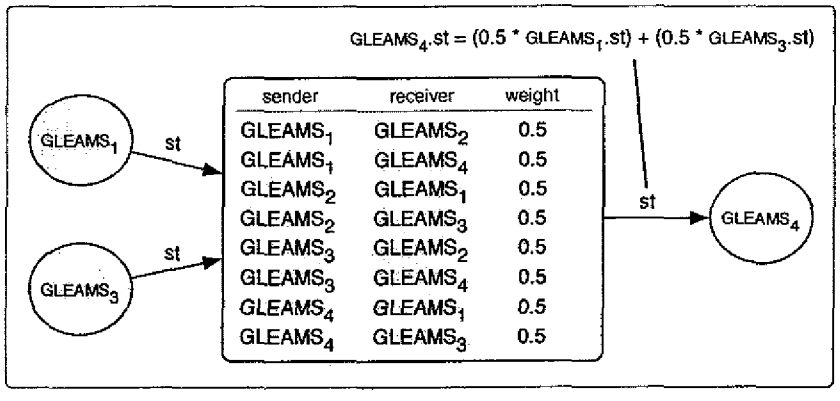

Figure 83. How values from multiple instances are combined.

The figure illustrates how the *st* value received by instance 4 is created. Instance 4 simulates the lower-left quadrant of the field, so the cardinal neighbors of this quadrant are simulated by instances 1 and 3, the top-left and bottom-right quadrants, respectively. Since there are two neighbors to the quadrant simulated by instance 4, the value received from each instance is scaled to half its value (the 0.5 in the figure) and is summed. This summed value represents the average water table head of the neighboring quadrants.

The data mapping can indicate any number of model instances. Consider the case where the field is divided into nine parts, as shown in Figure 84.

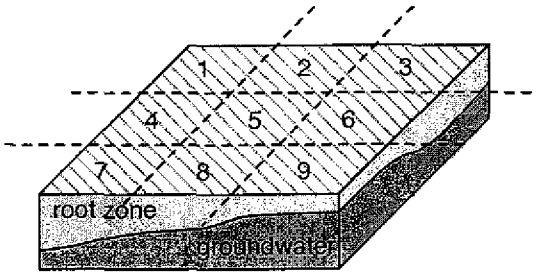

Figure 84. A field with nine parts.

Such a scenario is described by the data mapping in Figure 85. The corner instances communicate with their two neighbors, as in the previous data mapping, but the inner instances communicate with either three or four neighbors and the weightings are adjusted accordingly, either 0.33 or 0.25 respectively.

In these examples we wrote the data mappings by hand. The data mapping in Figure 85, which describes the communication between only nine instances, is more complex than the previous cases and writing data mappings for large numbers of instances would become tedious and error-prone. In such cases it is favorable to create the data mapping through the use of third-party software. This is explained in more detail in the case studies in Chapter 7.

```
GLEAMS Mapping - 9 Instances
Nine instances, arranged as a 3x3 grid
1   2   0.5
1   4   0.5
2   1   0.33
2   3   0.33
2   5   0.33
3   2   0.5
3   6   0.5
4   1   0.33
4   5   0.33
4   7   0.33
5   2   0.25
5   6   0.25
5   8   0.25
5   4   0.25
6   3   0.33
6   5   0.33
6   9   0.33
7   4   0.5
7   8   0.5
8   7   0.33
8   5   0.33
8   9   0.33
9   6   0.5
9   8   0.5
```

Figure 85. A data mapping indicating nine instances.

Summary

This chapter has introduced the Coupling Description Language and the coupling environment based on it within PCICouple. The actions of the CDL were explained, along with data mappings and techniques for keeping the models coordinated in time. Two examples were then presented that demonstrate the process of creating a coupling model in our approach. In the next chapter, we present the runtime system and explain how the actions described in coupling descriptions are carried out during execution of the coupled model.

CHAPTER VI

EXECUTING COUPLED MODELS

Introduction

In Chapter 4 we saw how PCIs and their associated coupling-ready model executables can be created through the use of PCICreate. We then saw in Chapter 5 how the behavior of a coupled model can be specified through the Coupling Description Language. This chapter presents the coupled model runtime system that is capable of executing these coupled models. An overview of the system is presented next, followed by an explanation of how coupling descriptions are compiled into scripts. A detailed description of the runtime system is then given.

Overview of the Runtime System

An overview of the runtime system is shown in Figure 86. Each shape in the figure represents an independently executing process, of which there are four kinds: *model instances* (circles), *couplers* (squares), *updaters* (diamonds), and *controllers* (house shape). The communication between these different processes is indicated by arrows in the figure (the controller also communicates with all of its associated processes, indicated by a dotted line in the figure, during startup and as described in the next section). Note that both updaters and model instances communicate only with couplers.

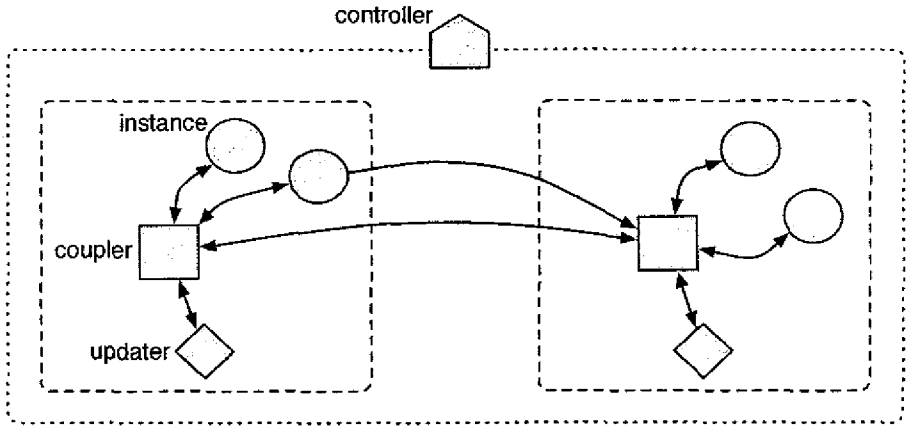

Figure 86. Overview of the runtime system.

Every instance is assigned to a specific coupler, indicated in the figure as a dashed line, and many instances may be assigned to the same coupler. An instance's coupler provides two services to the instance, value storage and value updating:

1. Couplers store the values of stored variables for their instances, and act as queues that collect values destined to their instances, holding them until their instances are ready for them.

2. To support Update Actions, couplers use their *updater* processes to carry out the execution of update functions. Each coupler has an updater process that can execute the built-in update functions and any custom functions added by the scientist. Couplers also apply data mappings, which is why instances do not communicate directly, but do so through couplers.

When executed on a parallel computer, the ratio of instances to couplers may have significant performance impacts since the couplers are providing potentially time-consuming services (function execution) to the instances, so this ratio is customizable by the scientist.

Compiling the Coupling Description

After a coupling has been described, the appropriate scripts for each model instance are created. As introduced in Chapter 4, a script is an ordered list of events that are to take place at the coupling points of a particular model instance; a separate script is created for each instance. There are two kinds of events: put and get. Put events indicate that the value of a variable needs to be used outside the model instance, and therefore must be sent by the instance. Get events indicate that the value of a variable needs to be set to a new value, and therefore the new value must be received by the instance. Note how actions are different from events: Actions are high-level constructs associated with a model, while events are low-level communications between specific instances of models. Each kind of action is decomposed into an event sequence which is said to be derived from that action. Some put and get events indicate that data (variable values) should be communicated, and others indicate that control requests should be communicated. Put and get events used for control are referred to as *put message* and *get message* events to distinguish them from put and get events that are used for data. For example, couplers do not make assumptions as to the state of each instance, so when an instance needs to receive a value (via a get event), it must request it from the coupler (through an initial put message event).

Send Actions

Send Actions indicate that the value of a variable should be sent from one group of instances to another group of instances, where the sending and receiving instances are determined by the associated data mapping. Put events are added to the script of each sending instance, one put event for each receiving instance. Put events specify which variable (value) at which coupling point should be sent, and which coupler should receive it.

Store Actions

Store Actions indicate that a stored variable should be created (or if it has already been created by an earlier activation of the Store Action, then its value is updated) and set to either a constant value or the current value of a model variable. In both cases, the instance must send a request to its coupler indicating that the stored variable should be created or updated. The value that the stored variable is set to can originate from three places and results in different events added to the instances' scripts in each case:

- If the stored variable is to be set to a constant value specified by the scientist, then a put message *store request* is added to the script of every instance of the model, and the coupler creates and stores the value.

- In the case where the stored variable is to be set to the current value of a local model variable, then a store request is added to the script of every instance of the model, immediately followed by a put event that sends the value of the local model variable to the coupler.

- In the case where the stored variable is to be set to the value of a model variable sent from another instance, no value is sent from the instance and hence no put event is added since the coupler will receive the value from the sending instance directly. In this case, only a store request is added, and it is only added to the scripts of the instances that are specified to receive the value from the sending instance (as indicated by the data mapping associated used by the sender).

In all cases, after the store request event (and after the put event if there is one) in the instance's script, a final get message event is added which causes the instance to wait until the coupler has fulfilled the request and sends a confirmation. This is important in the case where the a stored variable is to be set to the value

of a model variable sent from another instance, which may not have been received by the coupler by the time the coupler receives the store request.

Update Actions

Update Actions are used in the coupling description language to indicate that an update function should be applied to a set of variables, some of which may be local to the model in which the action is associated, and some of which may be values from other models. The local variables must be sent to the coupler that executes the function, and then received after the execution is complete (in order to reflect any changes in their values in the model). For this reason, one put event and one get event, for each local variable, is added to the script of each instance that will carry out the Update Action (the put events are added such that they occur before the get events). If all of the variables used in an Update Action are local to the model with which the action is associated, then all the instances of that model carry out the Update Action. If, however, some of the variables are sent from other model instances, then only the instances that receive those variables carry out the Update Action (if an instance does not receive a particular variable that is used in an Update Action, then that instance cannot perform the action). Along with the set of put and get events, there is also a put message *commence request* that is added after the put events which indicates to the coupler that the local variables have been sent and that the coupler should begin execution of the function after it has received any other necessary values. After the coupler has executed the update function, the resulting variable values are stored temporarily in the coupler and sent back to the instance in response to each of the instance's get events.

Example Compilation

To illustrate the compilation process, we show how the coupling description in Example 1 of Chapter 5 is compiled into a set of scripts. The coupling de-

scription includes two actions, a Send and an Update, as shown in Figure 62. The data mapping used by the Send Action indicates that there are two instances, and that instance 1 sends to instance 2. Therefore, there are two instances in the coupling, and a script is created for each. The script for instance 1 includes a single put event derived from the Send Action at Coupling Point B, as shown in Figure 87.

```
coupling point B: put hnew to instance 2's coupler
```

Figure 87. Script for instance 1.

The put event indicates that that the value of the *hnew* variable should be sent from Coupling Point B in instance 1, to instance 2's coupler. The script for instance 2 includes three events derived from the Update Action at Coupling Point A, as shown in Figure 88.

```
coupling point A: put stg to own coupler
coupling point A: put commence request to own coupler
coupling point A: get stg to own coupler
```

Figure 88. Script for instance 2.

The first event in the script for instance 2 sends the value of the *stg* variable to its coupler, and the second event sends a commence request to its coupler indicating that the coupler can start the function execution whenever it has received the value of the *hnew* variable from instance 1. The third event in the script causes instance 2 to stop and wait until its coupler has completed execution of the update function and returns the final value of the *stg* variable to the instance.

The result of this compilation process is a set of scripts, one for each instance of each model. These scripts though, simply instruct instances to send and receive variable values. How these values are communicated and transformed between instances is explained in the next section.

Operation of the Runtime System

The way in which a coupled model is started is described next, followed by an explanation of how couplers provide each of the two services listed earlier.

Starting a Coupled Model

When a coupling is executed in PCICouple, the description is compiled into scripts, and the controller is started. The controller oversees the execution of the coupled model and has control over starting and stopping all the other processes involved. The controller begins by starting each instance and each coupler, along with the updater for each coupler. The model instances contact the controller, and the controller sends a script to each instance. The couplers also contact the controller, and a copy of the coupling description is sent to each. Since there may potentially be a large number of instances and couplers contacting the controller simultaneously, the controller is multithreaded. After receiving its script, each instance begins its simulation. When a coupling point is reached by an instance, the coupling point's accessor subroutine is invoked. The subroutine iterates through the instance's script, sending and receiving values. The way in which these values are communicated is explained in the following two sections.

How Couplers Store Values

Couplers store values for instances in two ways in support of Send and Store Actions. When an instance performs a put event that was derived from a Send Action, the value is sent from the instance to the coupler that is assigned to the receiver instance (not to its own assigned coupler, unless both instances are assigned to the same coupler). The receiving coupler stores the value in a queue until it is needed, as shown in Figure 89. Instances never send values directly to other instances; the values are always sent to the coupler that is assigned to the receiving instance. In the figure, instance A sends a value to the coupler assigned to instance B. Put events always execute immediately (asynchronously).

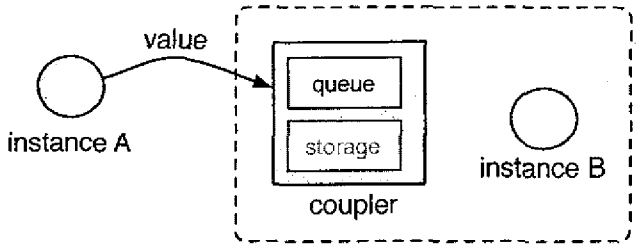

Figure 89. Instances send values to couplers only.

When an instance performs a put event that was derived from a Store Action then the value is sent from the instance to its assigned coupler, and the coupler stores the value, as shown in Figure 90 where instance B sends a value to its coupler, for storage.

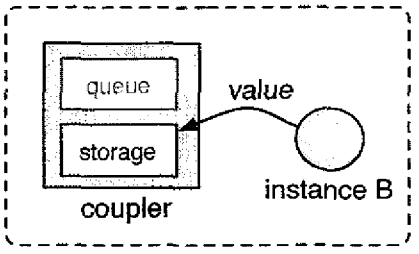

Figure 90. Couplers store values for their assigned instances.

If the value to be stored was sent from another instance, then when the coupler receives the store request, it moves the received value from its queue into its storage. When a stored variable is later used in Send or Update Actions, its value is sent by the coupler to another coupler or to its updater, respectively.

How Couplers Execute Update Functions

The coupler must be informed when an instance has reached an Update Action and is ready to execute an update function. This is the purpose of the commence request. When an instance sends a commence request (via a put message event), the request is sent to the coupler assigned to the instance. The instance then sends any of its local variable values to the coupler (via put events), and then waits to receive the new values from the coupler (blocking).

When a coupler receives a commence request (labeled 1 in Figure 91) from one of its assigned instances, it first checks to see if all the required variable values specified as arguments in the Update Action have been received into its queue. If they have not, the request is placed on a waiting list, and each time new values are added to the coupler's queue or storage, the waiting list is checked to see if any pending requests can be satisfied. Once all the necessary variable values have been received into the coupler's queue, the coupler then applies the associated data mappings (labeled 2 in Figure 91) in order to create the actual input values for the update function. These data mappings may cause values received from multiple instances to be combined in a weighted sum, or cause values to be scaled according to the weights specified in the data mapping.

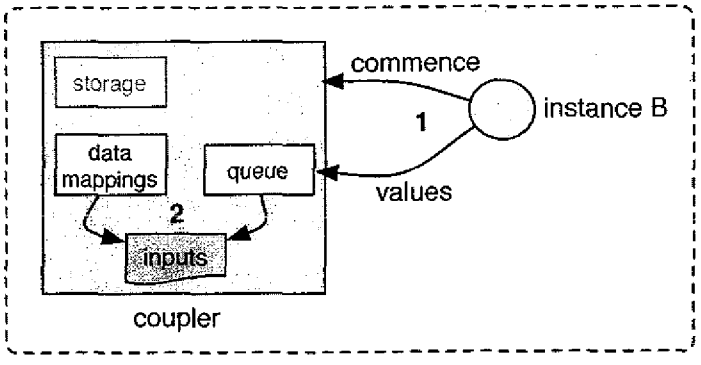

Figure 91. Function inputs are assembled by the coupler.

After all the inputs to the update function have been created, all the values are sent to the coupler's updater process. Each coupler has an independent process called an updater that handles the actual execution of update functions, and is written in the same language as the functions. The coupler cannot execute update functions itself because the functions may be written in a different programming language than the coupler. The coupler tells the updater which function to execute, and sends it the input values (labeled 1 in Figure 92).

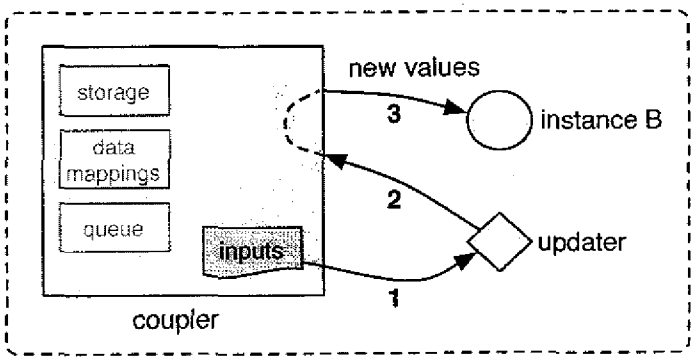

Figure 92. Couplers use updaters to apply update functions.

The updater applies the update function and sends the values (some of which may have changed) to the coupler (labeled 2 in Figure 92). The coupler then stores the updated values temporarily and then sends them to the instance that originated the commence request (labeled 3 in Figure 92). If a stored variable is used as an argument to an update function, then value of the stored variable is updated to reflect the value returned from the function. When a sent variable is used as an argument to an update function, the value returned by the updater is discarded since sent variables are considered to be read-only at the destination coupling point. After the instance receives the new values of its local variables, it resumes its simulation.

Summary

This chapter first explained how each kind of action is decomposed into scripts, and then described how the runtime system operates. Couplers play a central role in the execution of coupled models, providing two important services to model instances: storage of values, and updating of values. The next chapter presents a series of case studies that put everything together, from describing the coupling to executing the coupled model.

CHAPTER VII

CASE STUDIES

Introduction

This chapter presents three case studies that demonstrate the InCouple approach to creating coupled models. In the earlier examples in Chapter 5, the explanations focused on the process of creating the coupled models and the physical interactions and spatial distributions were simplified in order to focus on the presentation of the CDL. In these studies, the latter two of which conducted in collaboration with hydrologists, our approach to creating coupled models is demonstrated as it would be used in a scientific study. Although the interpretation of the coupled model results is beyond the scope of this work, we nonetheless show how coupled models can be created in a practical setting. The first study shows how an existing coupled model can be recreated using our approach, and evaluates our coupling in terms of the existing coupling. The second study investigates transport through the stream network of a watershed by coupling together many instances of the same model. The third case study investigates the interaction between rainfall-runoff and groundwater processes by coupling together two different models, each of which simulates one of these processes.

Case Study: Simulating Stream-Aquifer Interaction

Modeling is commonly used in hydrology to study both groundwater and surfacewater dynamics. These systems were originally modeled separately, resulting in the development of a variety of models that simulate only surfacewater or only groundwater. As scientists became more interested in comprehensive simulations, they began to develop ways to study the interaction between these two systems. In one such effort, Jobson and Harbaugh (1999) coupled the groundwater-flow model ModFlow (introduced in Example 1 of Chapter 5) to the surfacewater-flow model DAFlow. DAFlow (Jobson 1989) simulates the movement of water through a network of interconnected channels. It was developed by the U.S. Geological Society and consists of approximately 700 lines of Fortran code. An example of a physical system that can be simulated by DAFlow is illustrated in Figure 93.

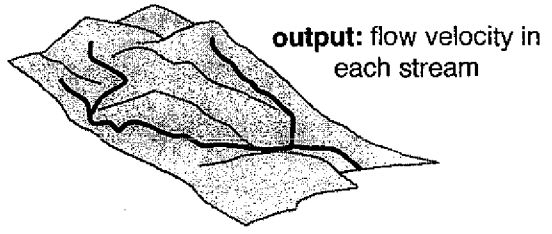

Figure 93. Illustration of the physical system simulated by DAFlow.

As a reference for evaluating our approach to model coupling, we compare the coupled model developed by Jobson and Harbaugh, the *reference coupling*, with a coupled model that we created using our approach, the *interface coupling*. The high-level design of the coupling common to both approaches is presented next, followed by a description of how it is implemented in the reference coupling and in the interface coupling. We then compare the couplings in terms of accuracy, efficiency, and design effort.

Coupled Model High-Level Design

The purpose of this coupling is to enable ModFlow to account for the presence of surfacewater in its simulation of groundwater, and to enable DAFlow to account for the presence of groundwater in its simulation of surfacewater. As illustrated previously in Figure 57, water flux between these systems occurs through the unsaturated zone as recharge (downward flux) and baseflow (upward flux). We use the term *seepage* in this study to describe the movement of water in both directions, where the sign of the seepage value indicates the direction of the movement of water). The seepage between the surfacewater channels and the groundwater aquifer must be calculated and then used to adjust the volume of water in each. We first explain how the seepage is calculated, and what variables in each model are used in the calculation. We then explain how the calculated seepage value is used to adjust the values of the relevant state of variables of each model.

Calculating Seepage

The calculation of the seepage is based on both the state of the surfacewater and the state of the groundwater, so we must establish an association between the channels in the stream network and groundwater beneath them. ModFlow abstracts the aquifer as a regular grid, and DAFlow abstracts the channel network as a network of branches connected by junctions. Each branch is divided into one or more segments where each segment is called a *subreach*. *Nodes* indicate the start and end of each subreach. The subreaches of each branch are delineated such that each subreach lies over a single grid cell. This is illustrated in Figure 94 (adapted from Jobson and Harbaugh 1999).

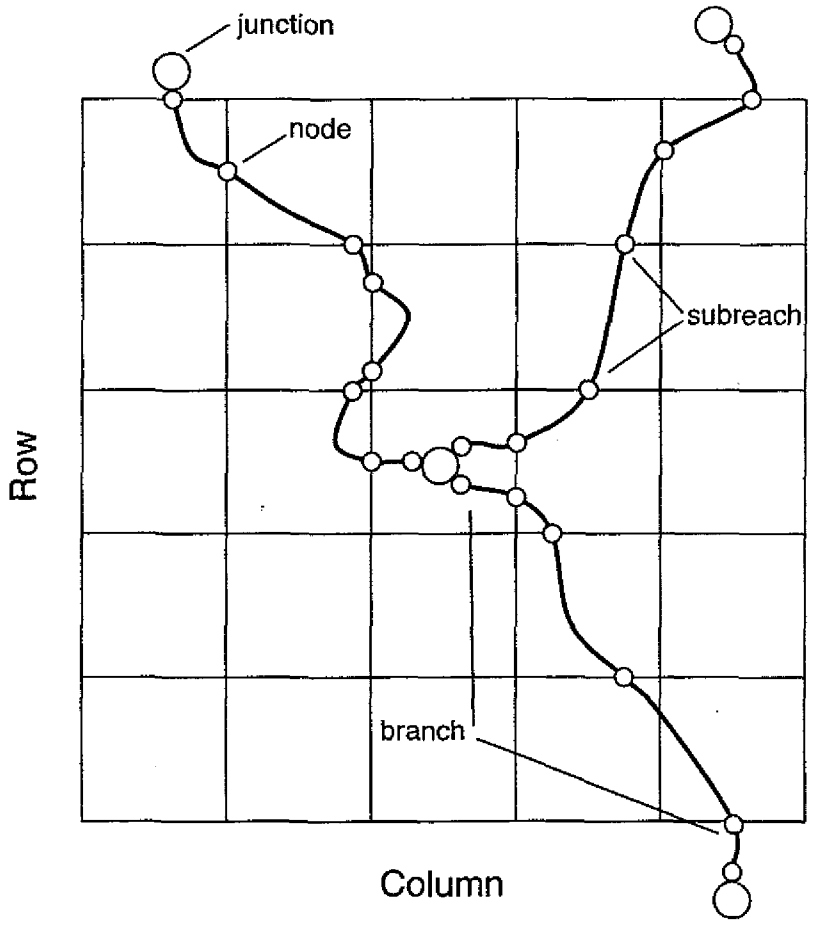

Figure 94. Each subreach is associated with a single grid cell.

The calculation of seepage therefore involves determining the seepage between each subreach and the grid cell beneath it. The calculation of the seepage between a subreach and a grid cell is given by the following equation:

$$S_{ep} = K_c\ L\ W\ (H_d\text{-}Y\text{-}B_e)/B_t$$

where Sep is the flow from the aquifer to the stream through the streambed, K_c is the hydraulic conductivity of the streambed, L is the length of the subreach, W is the average width of the subreach, H_d is the head of the aquifer, Y is the average depth of the subreach, B_e is the average elevation of the streambed, and B_t is the thickness of the streambed. These are illustrated in Figure 95 (adapted from Jobson and Harbaugh 1999).

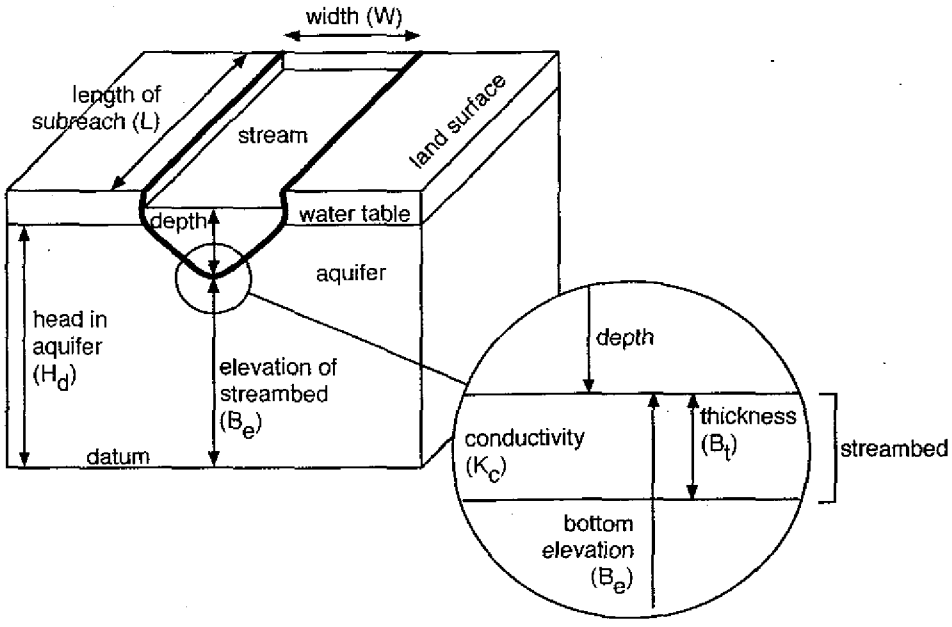

Figure 95. Physical quantities that influence seepage.

H_d is represented by the *hnew* variable in ModFlow, and L is represented by the *x* variable in DAFlow. The other variables though, K_c, W, Y, B_e, and B_t are not included in either model because these quantities are only necessary to simulate the seepage between surfacewater and groundwater, but not simulate either individually. Therefore, these additional characteristics of each subreach must either be calculated or supplied to the coupled model in addition to the typical model inputs. The variables W and Y are calculated from the following variables in DAFlow: the subreach velocity, *v*, two cross sectional area values, *ao* and *a1*, an exponent of area value, *a2*, and two equation coefficients related to the stream width, *w1* and *w2*. The terms B_e, B_t and K_c are constants supplied to the coupled model. After the seepage is calculated, it is used to adjust the state of each model.

The calculated seepage is used to adjust the volume of water in the aquifer through two variables in ModFlow, *rhs* and *hcof*. These variables are arrays in which each element is associated with a cell in the grid. They are terms that are used in the equation solved by ModFlow, but conceptually represent the introduc-

tion or removal of water from each cell. The seepage is converted into the appropriate units and is used to affect these variables.

The seepage is used to adjust the new flow at each junction, represented by DAFlow's *trb* variable, with units ft^3/s. Each junction in DAFlow may introduce water into a subreach, or divert water before it reaches the downstream subreach. Each element in the *trb* array represents the new flow at a different junction in the network. So, to add the seepage to a subreach is to add the seepage to the upstream junction of the subreach.

The seepage is calculated and used to adjust the state of each model periodically throughout their simulations. ModFlow uses a time step length that is generally much longer than the step length used in DAFlow, and as a result, the DAFlow model may need to simulate several smaller time steps for each longer ModFlow time step. In such a case, the seepage is calculated and used to adjust the state of each model once on each of the longer time steps.

We have explained how the seepage is calculated, which variables are needed from each model to perform the calculation, and which variables in each model are updated as a result of the calculation. We have also described how the models are spatially mapped and coordinated temporally. Next, we explain how this is implemented in the reference coupling, followed by an explanation of how it is implemented in the interface coupling.

Implementation of the Reference Coupling

The reference coupling was implemented using the monolithic approach discussed in Chapter 2. The DAFlow source code was divided into subroutines and inserted into the ModFlow model code to create a single model code. Additional source code was then added that calculates the seepage and updates the values of the relevant variables. Figure 96 (adapted from Jobson and Harbaugh

1999) shows how the DAFlow model code was decomposed and integrated into the ModFlow model code. In this figure, edges indicate which DAFlow subroutines are invoked at which locations within ModFlow.

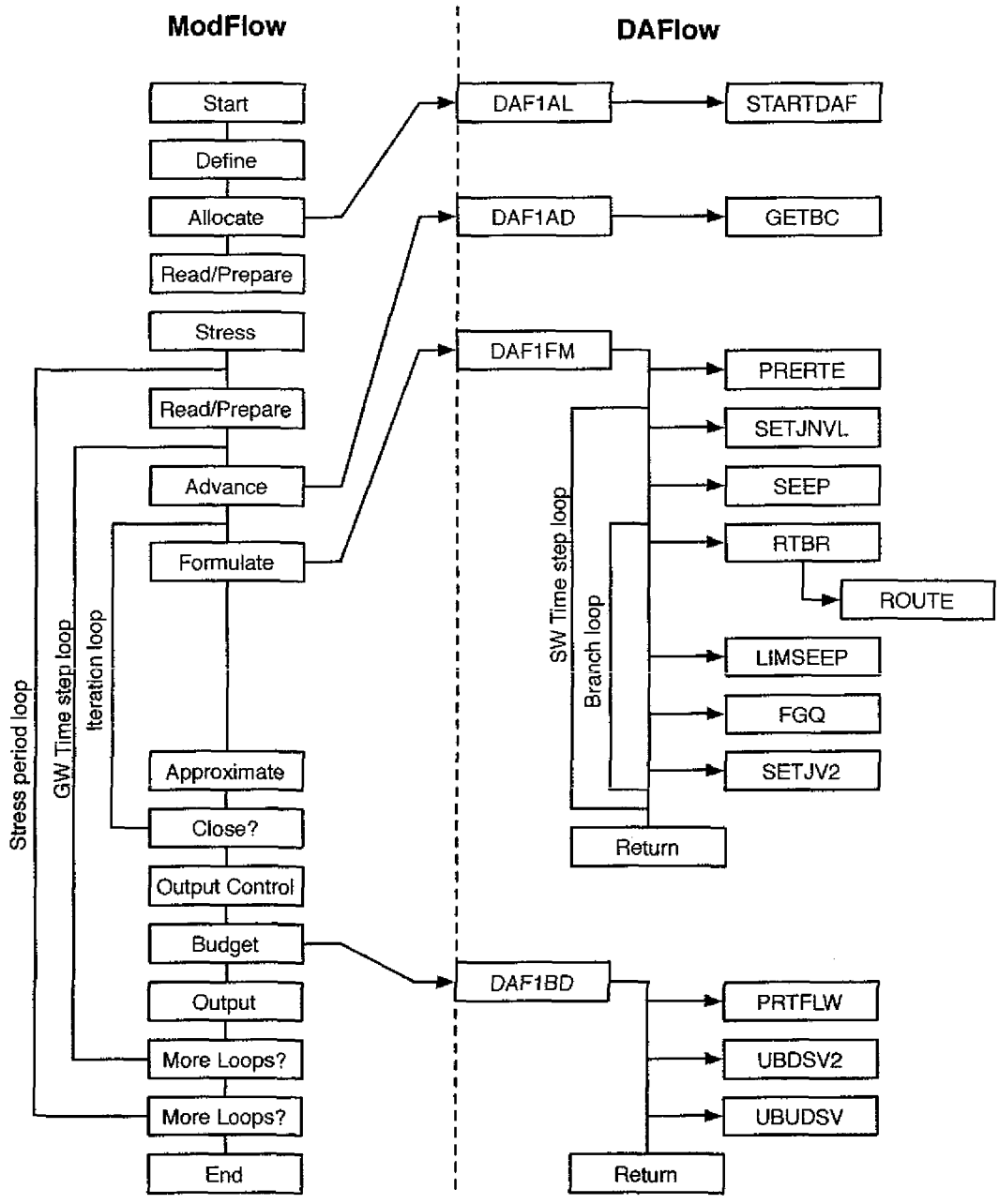

Figure 96. How the model codes were integrated, arrows indicate function calls.

The DAFlow model code was divided into four subroutines (their names are prefixed by "DAF"), each of which is a wrapper around one or more existing subroutines in DAFlow. On each time step in ModFlow, the groundwater flow equations are iteratively formulated and approximated. As part of the formulate step, ModFlow invokes the *DAF1FM* subroutine which carries out some number of DAFlow time steps. At the start of each of these time steps, the seepage is calculated in the *SEEP* subroutine called by *DAF1FM* and the value of the *trb* variable is adjusted accordingly. Then as part of the branch loop within each time step, the flow in each branch is simulated by the *ROUTE* subroutine. After each branch is routed, the *LIMSEEP* subroutine is invoked which limits the amount of seepage in case there is not sufficient water in the stream. Since the *DAF1FM* subroutine may be called several times on each time step of ModFlow, the state of DAFlow is saved in the *DAF1AD* subroutine and restored at the start of the *DAF1FM* subroutine, allowing sets of time steps to be re-simulated in DAFlow.

To calculate the seepage, the *SEEP* subroutine must know which stream channels are associated with which groundwater grid cells. This is specified by the scientist through an additional model input file which lists each subreach and which grid cell is beneath it. The additional physical characteristics of each subreach needed for the seepage calculation, streambed thickness, bottom elevation, and hydraulic conductivity, are also specified in this file for each subreach.

Since ModFlow typically uses time step lengths on the order of days or weeks, while DAFlow uses time steps on the order of hours or days, the *DAF1FM* subroutine is therefore capable of simulating several time steps and includes the time step loop. In this way, if ModFlow uses a time step length of m, and DAFlow uses time step length of d, where $d < m$, then m must be evenly divisible by d, and the number of DAFlow time steps simulated s, is equal to m / d.

Implementation of the Interface Coupling

The reference coupling was designed to fully integrate the DAFlow model into the ModFlow model in a general way so that it could be used in a variety of scenarios and support different dynamics between the models, including all three cases shown previously in Figure 58. Since we are interested in creating this coupling to specifically compare it to the reference model, we did not duplicate the capabilities of the reference coupling exactly, just enough to reproduce the results of one of the example applications included in the documentation for the reference coupling. We point out the simplifications made in our discussion and explain how each could alternatively be implemented exactly as in the reference model. The coupling description is shown in Figure 97. At the start of each time step, ModFlow sends the value of the *hnew* variable from Coupling Point A to Coupling Point D in DAFlow. Since there is only one instance of each model in this coupling, the default 1-to-1 data mapping can be used. The custom update function *calcSeepage* is applied at the destination coupling point and calculates the seepage. The subroutine is based on the *SEEP* subroutine used in the reference coupling and calculates three values, each of which is saved in a different stored variable. The seepage, represented as a flow rate, is stored in the *sep* stored variable, and the seepage, represented in terms of ModFlow 's cell volumes, is stored in the *rhs* and *hcof* stored variables. All three stored variables are initialized at Coupling Point C in DAFlow. The source code for the *calcSeepage* subroutine is shown in Figure 98.

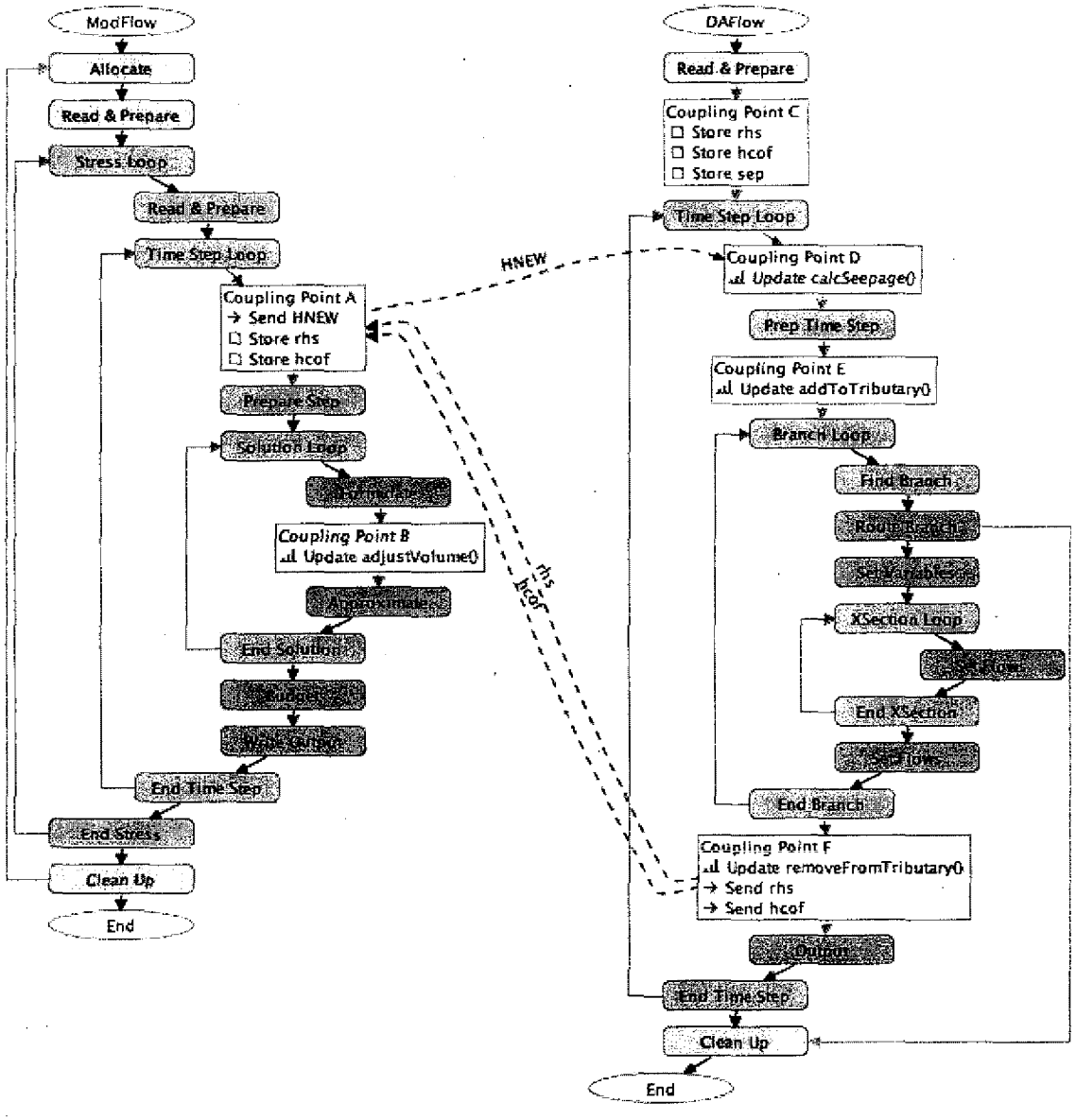

Figure 97. The coupling description as it appears in PCICouple.

In an effort to implement the interface coupling in a manner similar to the reference coupling, the mapping of grid cells to subreaches is embedded within the subroutine (the first loop in Figure 98). It could alternatively be accomplished using data mappings, or by reading the mapping in from a file as in the reference coupling.

```
    subroutine calcSeepage(instanceID,hnew,rhs,hcof,vin,x,ao,a1,a2,w1,w2,sep)

    double precision hnew(39,13,1)
    integer instanceID,nrw(15,13),ncl(15,13),n,i
    real    rhs(15,13),hcof(15,13),vin(15,13),x(15,13),ao(15,13)
    real    a1(15,13),a2(15,13),w1(15,13),w2(15,13),hd
    real    bel(15,13),bth(15,13),cnd(15,13),vol,q,w,dpt,ar
    real    sep(15),qstr(15,13),stage(15,13),cstr(15,13)

    n = 1
    do 10 i=1,15
        bel(i,n) = 46.00
        bth(i,n) = 1.00
        cnd(i,n) = 3.70E-04
        nrw(i,n) = i-1
        ncl(i,n) = 20
        rhs(i,n) = 0
        hcof(i,n) = 0
        sep(i) = 0
        qstr(i,n) = 0
        stage(i,n) = 0
        cstr(i,n) = 0
10  continue

    do 20 i=1,13
        vol = vin(i,n) / (x(i+1,n) - x(i,n))
        q = ((vol - ao(i,n)) / a1(i,n))**(1.0 / a2(i,n))
        w = w1(i,n) * (q**w2(i,n))
        dpt = vol / w
        hd = hnew( ncl(i+1,n), nrw(i+1,n), 1 )
        hd = hd-bel(i+1,n) - dpt
        ar = (w+2.0*hd) * (x(i+1,n)-x(i,n))
        cstr(i+1,n) = cnd(i+1,n) * ar / bth(i+1,n)
        sep(i+1) = cstr(i+1,n) * hd
        stage(i+1,n) = bel(i+1,n) + dpt
        qstr(i+1,n) = 0.0

        rhs(i+1,n) = rhs(i+1,n) - stage(i+1,n) * cstr(i+1,n) + qstr(i+1,n)
        hcof(i+1,n) = hcof(i+1,n) + cstr(i+1,n)
20  continue
    end
```

Figure 98. The source code for the calcSeepage subroutine.

The *calcSeepage* function has been simplified slightly such that some situations, such as when the surfacewater has dried out, are not accounted for. This function also assumes that there is only a single branch in the stream network. Additional modifications to the function could remove these limitations but were not necessary for our case study.

The calculated seepage value, *sep*, is added to the new tributary flow *trb* in DAFlow at Coupling Point E, before the branch loop. After the branch loop, the seepage is removed from the tributary flow because this value is used on the next iteration of the time step loop, and if not removed, the added seepage will be compounded on the next iteration. The source code for the *addToTributary* and *removeFromTributary* functions is shown in Figure 99.

```
    subroutine addToTributary(instanceID,trb,sep)
    integer instanceID,i
    real    trb(25,20),sep(15)
    do 10 i=1,15
       trb(i,1) = trb(i,1) + sep(i)
10  continue
    end

    subroutine removeFromTributary(instanceID,trb,sep)
    integer instanceID,i
    real    trb(25,20), sep(15)
    do 10 i=1,15
       trb(i,1) = trb(i,1) - sep(i)
10  continue
    end
```

Figure 99. The source code for the tributary functions.

We simplify the coupling in that the *LIMSEEP* is not performed, since it did not have an effect in the example application. This could easily be incorporated by adding another update function.

The seepage values that are calculated by the calcSeepage update function that are to be used to adjust the state of ModFlow, *rhs* and *hcof*, are sent from Coupling Point F in DAFlow and received at Coupling Point A in ModFlow. The data mapping used by the Send Action is the Reverse 1-to-1 mapping in which instance 2 sends to instance 1. Since the *rhs* and *hcof* values need to be used at Coupling Point B, they are stored when they are received at Coupling Point A. At Coupling Point B, the custom update function *adjVolume* is invoked

which adjusts the *rhs* and hcof variables based on the stored values that bear the same name. The source code for the *adjVolume* function is shown in Figure 100.

```
    subroutine adjustVolume(instanceID,rhs,hcof,seeprhs,seephcof)

    integer instanceID,i,n
    real    rhs(39,13,1),hcof(39,13,1),seeprhs(15),seephcof(15)
    real    bel(15,13),bth(15,13),cnd(15,13)
    integer nly(15,13),nrw(15,13),ncl(15,13)

    n = 1
    do 10 i=2,14
        bel(i,n) = 46.00
        bth(i,n) = 1.00
        cnd(i,n) = 3.70E-04
        nly(i,n) = 1
        nrw(i,n) = i-1
        ncl(i,n) = 20
10  continue

    do 20 i=1,13
        rhs( ncl(i+1,n), nrw(i+1,n), 1 ) =
  &         rhs( ncl(i+1,n), nrw(i+1,n), 1 ) + seeprhs(i+1)

        hcof( ncl(i+1,n), nrw(i+1,n), 1 ) =
  &         hcof( ncl(i+1,n), nrw(i+1,n), 1 ) - seephcof(i+1)
20  continue
    end
```

Figure 100. The source code of the adjVolume function.

Comparing the coupling description to the flow chart of the reference implementation in Figure 96, it can be seen that the reference implementation invokes the entire time step loop multiple times, while in the InCouple implementation, the models communicate within the DAFlow time step loop. This is acceptable because both models use the same time step length in the example application. If however, we wish to execute several time steps in DAFlow for each time step in ModFlow, the activation frequencies of the actions in DAFlow would have to be increased accordingly.

Note that since the interface coupling was implemented to closely follow the reference coupling, and the reference coupling was designed in for the mono-

lithic approach, it may not be the ideal way to design the interface coupling and that a different design that takes advantage of the data mapping capabilities of the CDL may be more ideal. Next we present the study site to which both coupled models are applied.

The Study Site

The user documentation for the reference coupling includes three example applications. We compare the reference coupling to the interface coupling as each is applied to the study site used in the first example. The example application site consists of an idealized unconfined aquifer with a stream flowing north to south as shown in Figure 101 (adapted from Jobson and Harbaugh 1999).

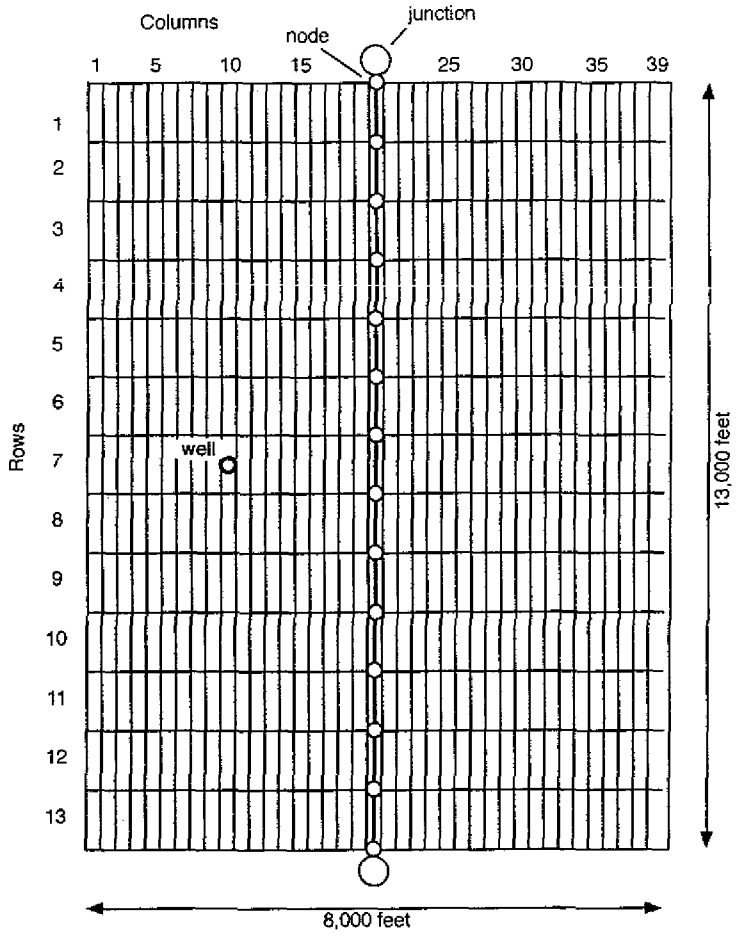

Figure 101. The schematic for example application 1.

The width of the aquifer perpendicular to the stream is 4,000 ft on each side, while the length parallel to the stream is 13,000 ft. The aquifer was represented in ModFlow by a grid with 39 columns and 13 rows. Each cell was 1000 ft long and 200 ft wide, except for cells in columns 1 and 39 which were 300 ft wide. The stream ran vertically through the center of column 20. The annual cycle was represented in Modflow by 24 stress periods each 15 days long, and each stress period was divided into two 7.5 day time steps. ModFlow is parameterized to introduce 1.5 feet of recharge evenly across the aquifer. The daily recharge rate has a sinusoidal distribution for the first 180 days, and is zero for the remaining 180 days. Additional parameters and assumptions are given in the user documentation.

Evaluation of the Interface Coupling

We evaluate the interface coupling in terms of the reference coupling in three respects: accuracy, efficiency, and effort. We reproduced the parameter input files for the reference coupling according to its documentation, and created the input files for each model in the interface coupling based on these input files.

The accuracy of the reference coupling was evaluated by the original authors (with respect to known analytical solutions) and was found to be acceptable. For this reason, the output from the reference coupling was used as a basis of comparison for the interface approach. After executing each coupled model, their output was found to be identical, indicating that the interface coupling can provide sufficient accuracy (at least in some cases). Figures 102 and 103 compare the output of the two coupled models. The simulated streamflow at node 14 as a function of time is compared in Figure 102.

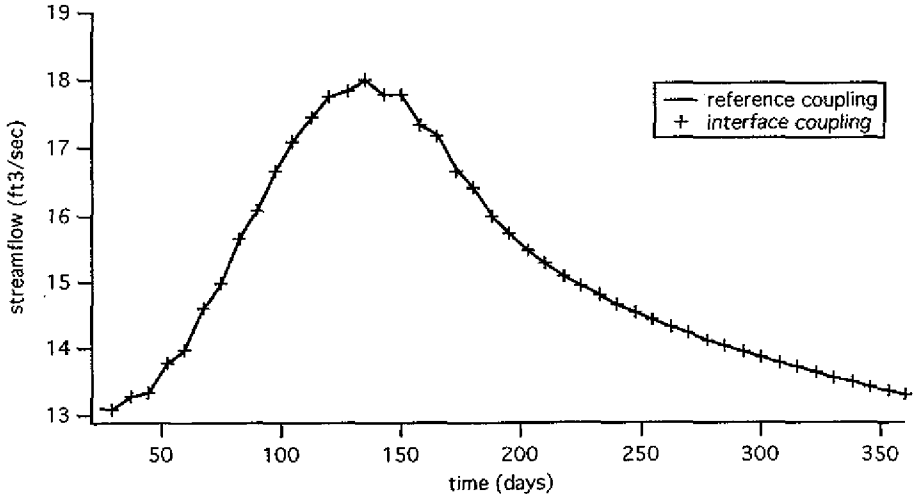

Figure 102. Simulated streamflow at node 14 for each coupling.

Profiles of the aquifer head in row 7 on the right side of the stream on day 177, near the end of the recharge period are compared in Figure 103.

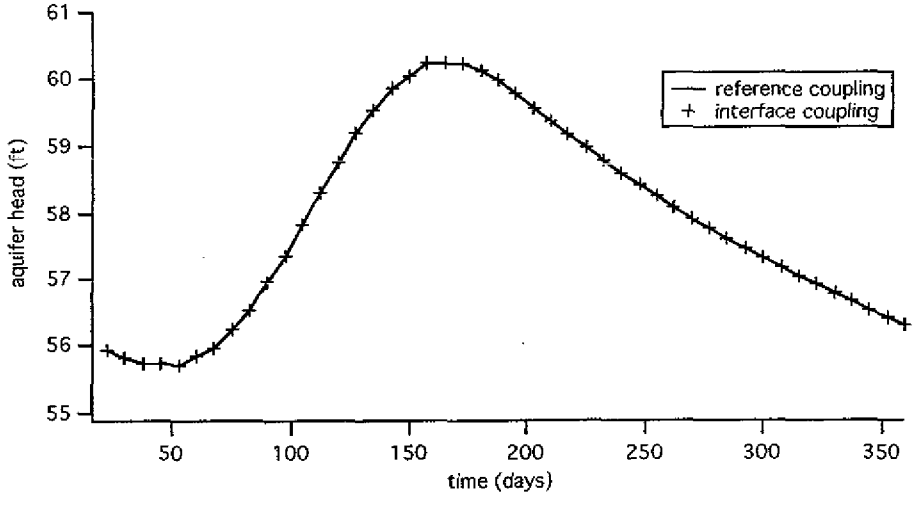

Figure 103. Comparison of aquifer head at a well located in row 7, column 10.

Efficiency was characterized by both time and bandwidth, although the bandwidth metric of the interface coupling does not have a corresponding metric with which it can be compared in the reference coupling. Time was the wall-clock time required by each coupling to complete its simulation. The reference model was timed using the unix time command, and the interface coupling execution time

was provided by comparing time stamps produced by PCICouple (this is because the interface coupling cannot be executed from a command line, so the time command could not be used). Each coupling was executed 10 times with no screen I/O on a 1.5 GHz PowerPC G4 computer. In the interface coupling, both models were executing on the same machine and shared the same coupler process. The bandwidth consumed by the interface coupling was determined from internal profiling performed by PCICouple. Table 6 shows the results of the coupled model executions.

Table 6. Time and bandwidth measurements of each coupled model.

	Execution Time (sec) mean / std dev	Bytes Sent & Received (MB)
Reference Coupling	1.03 / 0.04	-
Interface Coupling	70.4 / 3.5 run, 8.1 / 0.7 startup	34.2 sent, 20.2 received

The execution time of the reference coupling is much shorter, as would be expected since the reference coupling is a single process. 11.5% of the execution time in the interface coupling is due to the initial startup cost involved in starting the controller, coupler, updater, and two model instances. There is a penalty to be paid for the fast prototyping capability and clearly the coupling must be tuned for production runs. The bandwidth consumed by the interface coupling is a function of the size of the variables that are exchanged and the how often they are used in update functions, since each model variable used in an update function must be communicated a total of four times: from the model instance to the coupler and then to the updater, and back again. The overhead for the control messages is very low, as is the overhead for each message header.

The third metric of comparison is the design effort of the coupling which involves the difficulty involved in designing and implementing the coupled model. This is largely a subjective question as even qualitative measures such as development time are difficult to measure reliably. There are differences though in the

two strategies from the scientist's perspective. The interface approach did not require the scientist to edit the source code of the models once the PCIs had been created. This saved the scientist from the considerable overhead of learning the model code (and the language itself in which the model is written). Instead, s/he worked only in terms of the model PCIs.

Case Study: Watershed-wide Surfacewater Transport[3]

Water quality is a rising concern across the country and throughout the world and a common use of modeling in the field of hydrology is in the assessment of water quality in which scientists study how pollutants are transported through surfacewater and groundwater systems. Transport models can also be used to study how naturally-occurring nutrients move through water systems. One site at which a good deal of investigation has been conducted is the H. J. Andrews Experimental Forest. The Andrews Forest is situated in the western Cascade Range of Oregon in the 15,800-acre drainage basin of Lookout Creek, a tributary of Blue River and the McKenzie River. Elevation ranges from 1,350 feet to 5,340 feet. The Andrews Forest is broadly representative of the rugged mountainous landscape of the Pacific Northwest and contains excellent examples of the region's forest, wildlife and stream ecosystems. Figure 104 shows the gaging station at Mack Creek where stream data is collected.

Figure 104. Mack Creek gaging station in the Andrews Experimental Forest.

[3] in collaboration with Roy Haggerty, Oregon State University

Long-term field experiments and measurement programs have focused on climate dynamics, streamflow, water quality, and vegetation succession. Currently researchers are working to develop concepts and tools needed to predict effects of natural disturbance, land use, and climate change on ecosystem structure, function, and species composition.

In a recent study (Gooseff et al. 2004), researchers conducted *tracer tests* in four streams in the Andrews Forest and compared the empirical results with simulation results. In tracer tests, a measured concentration of a specific substance is introduced into a stream, and then the amount of concentration that passes through a downstream point in that stream is measured over time. The resulting concentration timeseries is called a *breakthrough curve*. The study compared the accuracy of two models with respect to how well they could reproduce the tracer tests in each of the streams. Both the models used, STAMMT-L and OTIS, simulate transport through a single, homogeneous steam, requiring that each of the four streams be simulated individually. If, however, the researchers wanted to simulate transport through the entire stream network within the Andrews Forest, it would be a time-consuming and error-prone process to simulate each of the 347 stream segments individually. In a related study (Lindgren, Destouni, and Miller, 2004), scientists avoided the individual simulations by creating a new model that is capable of simulating transport through an entire watershed. The task of developing a new model though, is time-consuming, difficult, and requires a significant upfront investment of resources. In our approach, we reuse existing models to quickly create the watershed-wide model. This coupled model could then be coupled to another model.

In this case study we show how a coupled model can be created by coupling together many instances of the STAMMT-L model to support the simulation of transport through a complete watershed. We also show how the preparation of data mappings can be automated through the use of a Geographic Information

System (GIS). An overview of the coupling is described next. Preparation of the data mapping is then explained, followed by the coupling description. A discussion of initial results is then presented.

Coupled Model Overview

STAMMT-L simulates the transport of a substance along a single, one-dimensional stream. As input, the model is given a timeseries of concentrations that describe the introduction of the substance into the upstream end of a stream. As output, the model produces a breakthrough curve. This is illustrated in Figure 105.

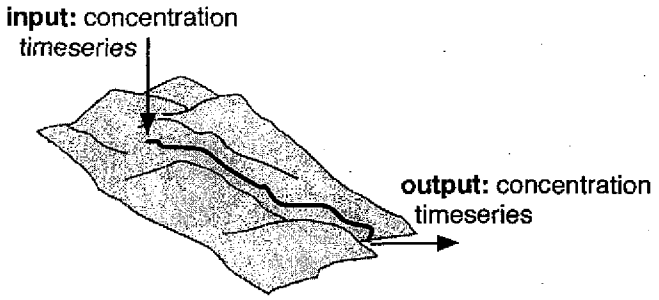

Figure 105. Illustration of the physical system simulated by STAMMT-L.

The time step length is typically on the order of minutes, and the period of time simulated varies from weeks to many years.

Since STAMMT-L is a lumped-parameter model, each stream in the watershed is simulated by a different instance of STAMMT-L. The input concentration timeseries for the outlying streams (those with no upstream stream) are specified in the model input files, and the input concentration timeseries for the inner streams (those that are connected to one or more upstream streams) use the sum of the output breakthrough curves of its connected upstream streams. All the instances, except for the instance simulating the downstream-most stream, send their output timeseries to another instance. This is shown in Figure 106. This is conceptually similar to a dataflow style of computation, where each in-

stance can be thought of as transforming a set of inputs to create a set of outputs.

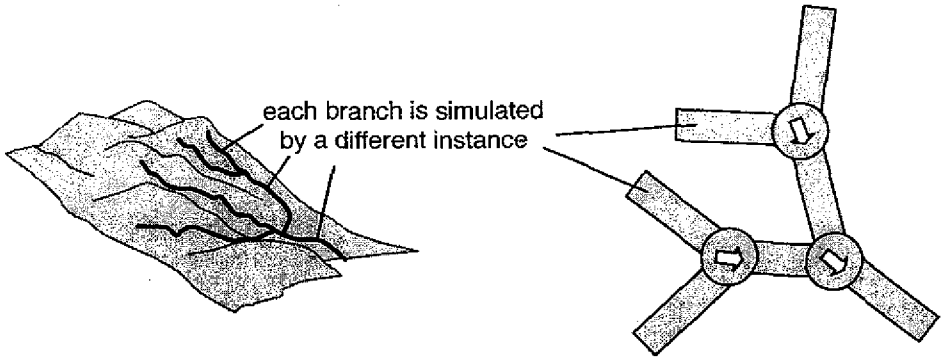

Figure 106. The physical stream network and its abstraction.

Each of the streams has different characteristics, which must be reflected in the simulations. In Example 2 in Chapter 5, each instance was told to use a different input file (via Update Actions) and the scientist prepared one input file for each instance. In this case, rather than create an individual input file for each instance, we use a different approach, to demonstrate the flexibility of update functions, in which all the instances read the same input file and the variables that describe the unique characteristics of a stream are changed individually within the model via Update Actions. We wrote a custom update function called *setParameterValues* that accepts a set of variables as input and sets each one to a value that is read from an input file called a *value list*. The value list indicates what value should be assigned to each variable for each instance. The next section describes how the value lists used by this update function are automatically generated through the use of a GIS.

The Coupled Model Inputs

A typical input file for STAMMT-L is shown in Figure 107. The model input parameter values are shown on the left in the figure, and a short description of each is given on the right.

```
100                ! ni
50                 ! nm
500                ! number of time steps (nt)
5                  ! nx (reverse sim only)
306.400D+00        ! branch length (L)
300.000D+00        ! distance to where conc is observed (xx)
0.102              ! dispersion (DL)
0.118              ! water velocity (vx)
1.00D+00           ! Rm
15.02D+00          ! btot
1.13D+00           ! dilute
1                  ! Ltime
0                  ! Lz
1000               ! start time in seconds (Tmin)
20000              ! end time in seconds (Tmax)
0                  ! ic
1                  ! bc_form
3                  ! bc_type
11                 ! how long the conc is added to branch (tp)
0.0D+00            ! disc
5000               ! kmax
1.0D-04            ! relerr should be around D-05
1                  ! opt
0                  ! ocm
1                  ! lcom
8                  ! mass_xfer_type
1.30               ! ln(rate) -26.4922D+00
7.05D-08           ! vary
2.50D-05           ! varv
```

Figure 107. The input file used by STAMMT-L.

The parameters that are unique to each instance are the ones that describe the physical characteristics of each stream, such as its length. It is these parameters that must be included in the value list and given to each instance. Table 7 shows which parameters are the same for all the instances, and which ones are instance-specific. Collecting these ten parameters for possibly hundreds of streams in a watershed is typically prohibitively expensive and rarely done in practice. For this reason, rather than measure these parameters for every stream in the watershed, they are only measured for a representative group of streams from which the parameters for all the streams can be estimated.

Table 7. The input parameters for STAMMT-L.

General parameters that are the same for all instances	Physical characteristic parameters that are unique for each instance
ni - number of input concentrations	vx - velocity
nt - number of simulation times	alphaL - longitudinal dispersivity
Rm - retardation factor for mobile zone	btot - total capacity coefficient
Ltime - flag for determining solution times	dilute - dilution factor for concentrations
Lz - flag for time step increment	par1 - vary
Tmin - minimum simulation time	par2 - vary
Tmax - maximum simulation time	par3 - vary
ic - initial conditions flag	bc_form - boundary condition form
bc_type - boundary condition flag	length - length of stream
tp - duration of concentration pulse	xx - observation point
disc - maximum discontinuity in laplace	Non-physical characteristic parameters that are unique for each instance
kmax - maximum number of iterations of laplace	
relerr - relative error desired for laplace	outconc - output filename for concentations
opt - flag for optimization	outmass - output filename for mass
ocm - flag for optimization	
lcom - flag for optimization	
massxfertype - diffusion coefficient flag	

In Spring of 2003, the parameters shown on the top-right in Table 7 were collected (Ninnemann 2004) for each general classification of stream size, or *order*. Lower order streams are smaller, while higher order streams are larger. The collected parameters are shown in Table 8.

Table 8. Representative properties of streams of different sizes.

Order	vx	alphaL	Btot	a_min	a_max	slope	dilute
1	0.05	0.49	12	6.9d-8	4.0d-5	-2.0	1.0
2	0.053	0.507	22	7.05d-8	2.5d-5	-2.0	1.12
3	0.179	1.09	176.5	4.6d-8	1.25d-4	-1.37	1.68
4	0.45	1.28	80.2	2.0d-7	2.58d-4	-1.34	2.72
5	0.57	1.36	156	1.9d-7	8.1d-4	-1.31	3.18

The parameters listed in the table can be used to determine the values for the upper 7 parameters listed on the right of Table 7, out of the 12 instance-specific parameters. Two of the remaining 5 parameters are the output filenames used by each instance (since all the instances can't write to the same file). Each instance uses the names Xoutconc.txt and Xoutmass.txt, where the X is replaced with the instance number, resulting in a unique set of output filenames for each instance (these are set via the *makeUnique* update function, not the *setParameterValues* function). The instance-unique values for the remaining 3 parameters are taken from a GIS. A great deal of spatial data about the Andrews Forest is available on-line[4], including both geographic and ecological information. The stream network and forest boundary data can be downloaded from the internet and imported into the ArcMap GIS, shown in Figure 108.

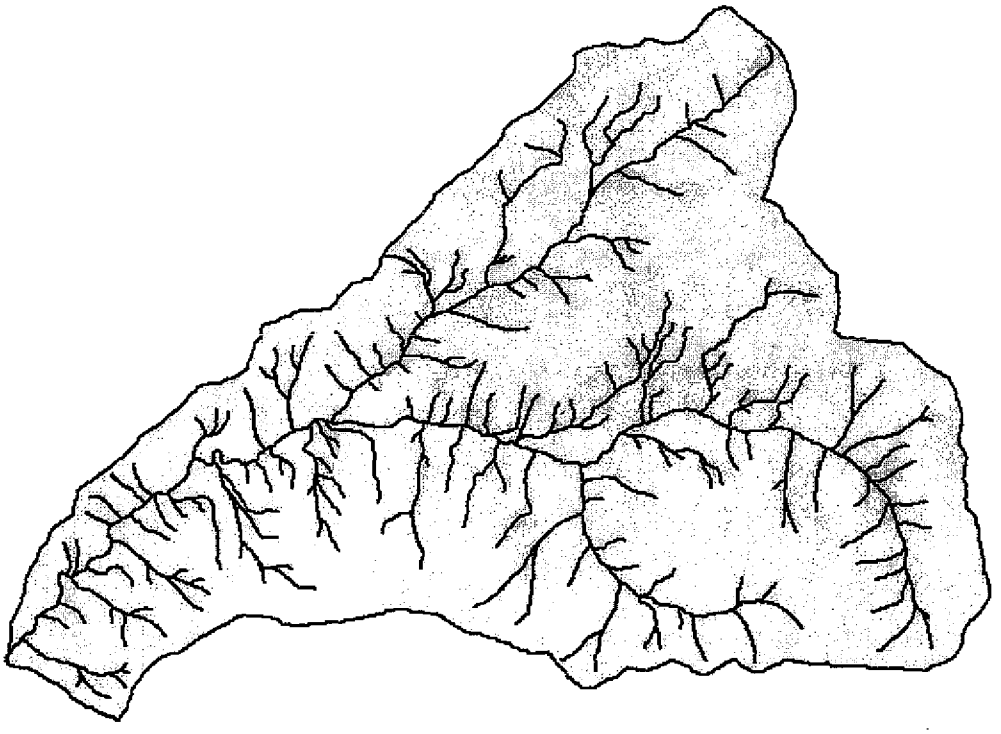

Figure 108. The stream network and forest boundary in ArcMap.

[4] http://www.fsl.orst.edu/lter/

In a GIS, each spatial unit (line, polygon, etc.) has a number of *attributes* associated with it such as its length or area. These attributes are stored in *attribute tables* in ArcMap. The attribute table included with the downloaded spatial data included the *length_* and *order_* of each stream. Since the order of each stream is present in the attribute table, the appropriate values for these 7 parameters from Table 7 can be easily matched to each stream.

The remaining 3 instance-unique parameters can be determined from the attribute table. The *xx* parameter is the *observation point*, which is the distance from the point where the concentration is introduced into the stream, to the point where the breakthrough curve is calculated. Since we want to calculate the breakthrough curve at the very end of each stream (so that it can be used as input for the downstream stream), the *xx* parameter is set equal to the length of each stream as given in the attribute table. The user manual for STAMMT-L suggests that the *length* parameter be set equal to twice the value of *xx*. The final instance-specific parameter is *bc_form*, which sets the kind of upstream boundary condition to use in the simulation. Three forms are supported by the model, but two are used in this coupling. The streams along the edges of the network, that have no upstream stream, should set the *bc_form* equal to 2, indicating that the model should introduce a concentration pulse as the input concentration to these streams, while the inner streams should set the *bc_form* equal to 3, indicating that the model should read a breakthrough curve from a file and use that as the input concentration. In the coupling though, the instances that have *bc_form* set to 3 will initially read in a dummy input file, and then set the variable equal to the breakthrough curve it receives from the upstream streams. Now that all the parameters for all the instances have been identified, the next step is to create the value list input file that is used by the *setParameterValues* update function.

The value list input file was created through the use of a custom ArcMap script that operates on the attribute table. Since the attribute table only describes

the length of each stream and how they are connected, the additional attributes of each stream that are unique to each instance (*vx*, *alphaL*, etc.) must be added to the table. A new attribute was added to the attribute table for each parameter, and its value for each spatial unit was set according to the data in Table 7. For example, to set the *bc_form* attribute for all the spatial units, the user would tell ArcMap to set the *bc_form* equal to 2 for all the units that are order 1, and set the *bc_form* equal to 3 for all the units that are of order greater than 1. This way the values for all the additional attributes can be quickly set. The extended attribute table, along with all the correct values for each stream, is shown in Figure 109.

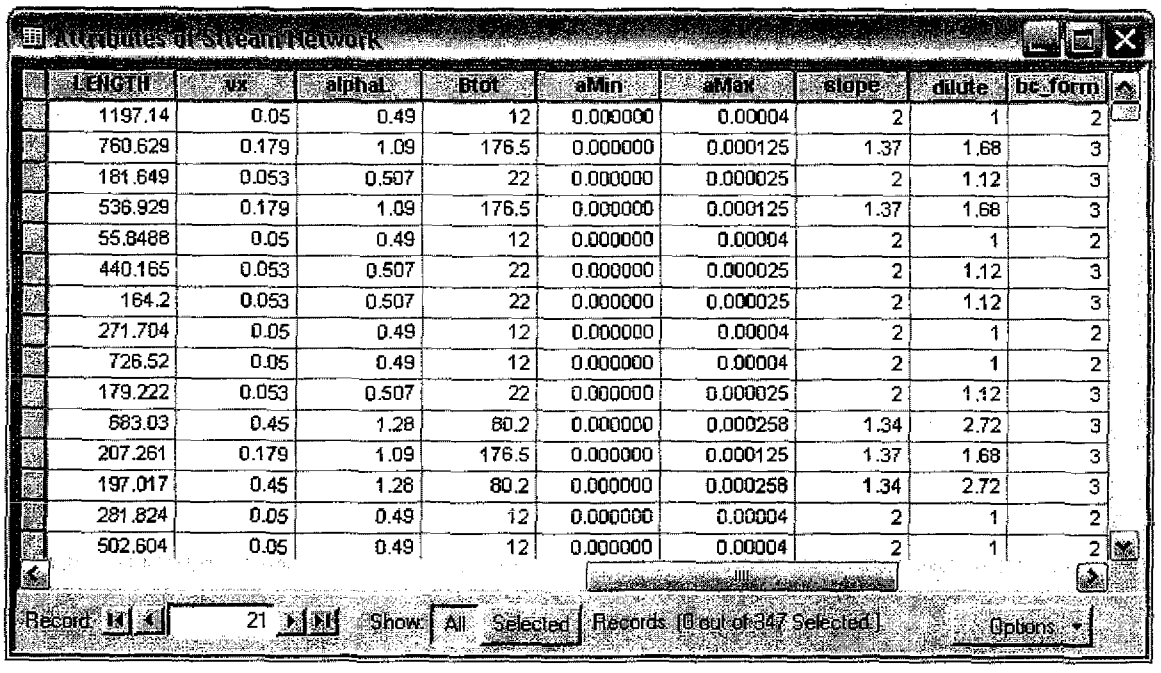

Figure 109. The extended attribute table for the stream network in ArcMap.

All of the attributes visible in the figure were added to the original data except for the *length* attribute. By doing this, all the instance-specific parameters for every instance are listed in the table, so creating the value list input file is simply a matter of translating the table into the value list input file format. This can be easily accomplished through the use of the scripting capability of ArcMap. Figure 110 shows a script written in Visual Basic (ArcMap's scripting language) that steps

through each stream in the attribute table and adds the necessary lines to a new value list file.

```
Public Sub createValueList()
  Dim pMxDoc As IMxDocument
  Dim pMap As IMap
  Dim FeatureClassTable As ITable
  Dim pRow As IRow
  Dim pFeatureLayer As IFeatureLayer
  Dim pFeatureSelection As IFeatureSelection
  Dim items As IEnumIDs
  Dim pLength,pOrder,pVX,pBtot,pDisp,pDilute,pSlope,pAMin,pAMax,pObsPoint As Double
  Dim rowCount, pBCForm As Integer
  Dim pOutMass, pOutConc As String

  Set pMxDoc = Application.Document
  Set pMap = pMxDoc.FocusMap
  Set pActiveView = pMap

  If Not TypeOf pMap.Layer(0) Is IFeatureLayer Then Exit Sub
  Set pFeatureLayer = pMap.Layer(0)
  Set pFeatureSelection = pFeatureLayer
  Set FeatureClassTable = pFeatureLayer.FeatureClass
  Set items = pFeatureSelection.SelectionSet.IDs

  rowCount = pFeatureSelection.SelectionSet.Count
  Open "value_list" & rowCount & ".txt" For Output As #1
  headerLine = Format(Now, "mm/dd/yyyy")
  Print #1, "# Created by ArcMap on " & headerLine

  items.Reset
  For i = 1 To rowCount
    Set pRow = FeatureClassTable.GetRow(items.Next)
    thisID = pRow.Value(pRow.Fields.FindField("FNODE_"))
    pLength = pRow.Value(pRow.Fields.FindField("Length"))
    pVX = pRow.Value(pRow.Fields.FindField("vx"))
    pBtot = pRow.Value(pRow.Fields.FindField("Btot"))
    pDisp = pRow.Value(pRow.Fields.FindField("alphaL"))
    pDilute = pRow.Value(pRow.Fields.FindField("dilute"))
    pSlope = pRow.Value(pRow.Fields.FindField("slope"))
    pAMin = pRow.Value(pRow.Fields.FindField("aMin"))
    pAMax = pRow.Value(pRow.Fields.FindField("aMax"))
    pBCForm = pRow.Value(pRow.Fields.FindField("bc_form"))
    Print #1, thisID & " " & (pLength * 2) & " " & pVX & " " & pBtot & " " & pDisp & "
" & pDilute & " " & pSlope & " " & pAMin & " " & pAMax & " " & pBCForm & " " & pLength
  Next i
  Close #1
End Sub
```

Figure 110. Script used to generate the value list.

The parameter list generated by the above ArcMap script is shown in Figure 111. The list includes parameter values for 10 variables, for 17 instances.

```
# Created by ArcMap on 04/20/2006
 1  2394.274  0.050   12.0  0.490  1.00  2.00  0.0000000690  0.000040  2  1197.137
21  1521.257  0.179  176.5  1.090  1.68  1.37  0.0000000460  0.000125  3  760.6287
 4  2049.115  0.050   12.0  0.490  1.00  2.00  0.0000000690  0.000040  2  1024.557
11  1838.221  0.053   22.0  0.507  1.12  2.00  0.0000000705  0.000025  3  919.1105
18  1099.461  0.053   22.0  0.507  1.12  2.00  0.0000000705  0.000025  3  549.7303
14  801.4677  0.053   22.0  0.507  1.12  2.00  0.0000000705  0.000025  3  400.7339
 3  2081.622  0.050   12.0  0.490  1.00  2.00  0.0000000690  0.000040  2  1040.811
10  523.3697  0.053   22.0  0.507  1.12  2.00  0.0000000705  0.000025  3  261.6848
 6  617.5225  0.050   12.0  0.490  1.00  2.00  0.0000000690  0.000040  2  308.7613
 5  1990.454  0.050   12.0  0.490  1.00  2.00  0.0000000690  0.000040  2  995.2269
19  2019.725  0.050   12.0  0.490  1.00  2.00  0.0000000690  0.000040  2  1009.862
 2  936.2112  0.050   12.0  0.490  1.00  2.00  0.0000000690  0.000040  2  468.1056
 7  939.5913  0.053   22.0  0.507  1.12  2.00  0.0000000705  0.000025  3  469.7957
 9  190.3888  0.053   22.0  0.507  1.12  2.00  0.0000000705  0.000025  3  95.19439
12  885.7083  0.050   12.0  0.490  1.00  2.00  0.0000000690  0.000040  2  442.8541
16  849.5131  0.053   22.0  0.507  1.12  2.00  0.0000000705  0.000025  3  424.7565
20  1803.333  0.050   12.0  0.490  1.00  2.00  0.0000000690  0.000040  2  901.6663
```

Figure 111. The generated parameter list (showing values for only 17 instances).

This section explained how a GIS can be used to automate the process of creating value lists that assign unique characteristics to individual instances. The next section describes the coupling description, and how these value lists are used therein.

The Coupling Description

As described earlier, each instance must receive an input concentration timeseries from the instances that simulate the upstream streams (if any), and then send their output breakthrough curves to the instance that simulates the downstream stream (if any). The input timeseries is stored in the *cdi* variable, and the output breakthrough curve is stored in the *c* variable. Both are arrays of type real. STAMMT-L does not use a specific system of units and the input values used are assumed to be consistent with each other. The exchange of the timeseries variables is accomplished through a Send Action and an Update Action, as shown in the coupling description in Figure 112.

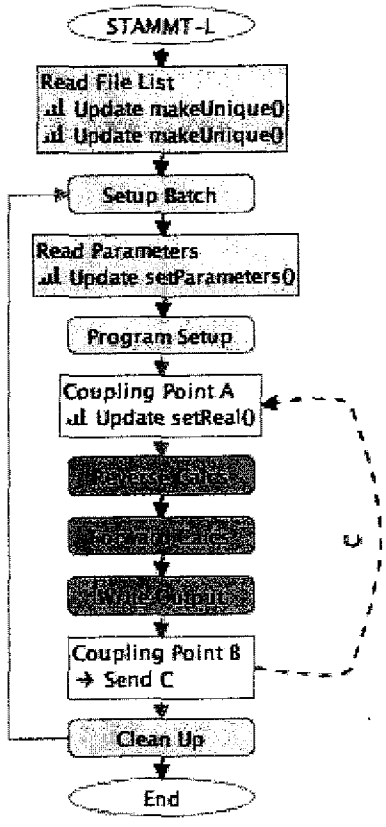

Figure 112. The coupling description.

The Send Action sends the *c* variable from Coupling Point B to Coupling Point A, according to the data mapping discussed next. This value is then used at Coupling Point A in an Update Action that applies the *setReal* function to set the value of the input timeseries variable, *cdi,* equal to the value of the received output timeseries variable, *c*. Both actions occur at an activation frequency of 1. Instance-unique values are assigned to the instances via the Update Action at the "Read Parameters" block which applies the custom function *setParameterValues*. The update actions at the "Read File List" block apply the makeUnique update function to the *outconc* and *outmass* variables which store the output filenames.

The Send Action uses a data mapping to describe which instances should send the *c* variable, and which should receive it. In the examples thus far, these

mappings have been short and simple, but in this case, we must describe the topology of hundreds of streams, a task which can be automated through the use of a GIS, in much the same way as the value lists were created earlier.

The attribute table shown in Figure 109 describes the topology of the stream network through three attributes: *objectid*, *tnode_* and *fnode_* (not visible in the figure). For each stream in the table, the *objectid* is a unique number assigned to each stream (pre-assigned in the downloaded dataset), and the *fnode_* and *tnode_* are the object IDs of the upstream and downstream streams, respectively. This information is precisely what is needed to create the data mapping, since the data mapping is what describes the topology of (communication between) instances in the coupled model. The script in Figure 113 creates the data mapping input file using these attributes.

```
Public Sub createDataMapping()
  Dim pMxDoc As IMxDocument, pMap As IMap, pRow As IRow, items As IEnumIDs
  Dim FeatureClassTable As ITable, pFeatureLayer As IFeatureLayer
  Dim pFeatureSelection As IFeatureSelection
  Set pMxDoc = Application.Document
  Set pMap = pMxDoc.FocusMap
  Set pActiveView = pMap
  If Not TypeOf pMap.Layer(0) Is IFeatureLayer Then Exit Sub
  Set pFeatureLayer = pMap.Layer(0)
  Set pFeatureSelection = pFeatureLayer
  Set FeatureClassTable = pFeatureLayer.FeatureClass
  Set items = pFeatureSelection.SelectionSet.IDs
  rowCount = pFeatureSelection.SelectionSet.Count
  Open "data_map" & rowCount & ".txt" For Output As #1
  headerLine = Format(Now, "mm/dd/yyyy")
  Print #1, "# Created by ArcMap on " & headerLine

  items.Reset
  For i = 1 To rowCount
    Set pRow = FeatureClassTable.GetRow(items.Next)
    modelID = pRow.Value(pRow.Fields.FindField("Model"))
    thisID = pRow.Value(pRow.Fields.FindField("FNODE_"))
    nextID = pRow.Value(pRow.Fields.FindField("TNODE_"))
    Print #1, modelID & " " & nextID & " " & thisID & " " & "1.0"
  Next i
  Close #1
End Sub
```

Figure 113. The ArcMap script used to generate the data mapping.

Generation of the data mapping is the final step in preparing the coupling description. The data mapping generated by the script in Figure 113, for 17 instances, is shown in Figure 114.

```
STAMMT-L 17 Instance Mapping
created by ArcMap on 04/20/2006
1   3  1.0
2   3  1.0
3   5  1.0
4   5  1.0
5   7  1.0
6   7  1.0
7   9  1.0
8   9  1.0
9  11  1.0
10 11  1.0
11 12  1.0
13 16  1.0
15 16  1.0
16 17  1.0
14 17  1.0
17 12  1.0
```

Figure 114. The data mapping used in this case study (for only 17 streams).

The model can be executed via PCICouple's execute command. The result of the coupled model run is a set of output breakthrough curve files, one for each instance in the coupling. These files are typically analyzed through third-party software such as MatLab. It would be easy to extend the implementation to support online program monitoring. Since all exchanged values are passed through couplers, these couplers could forward some of these values, as specified by the scientist, to a visualization user interface within PCICouple. In the next section, we present preliminary results of executing the coupled model.

Results[5]

Although a complete analysis of the coupled model outputs is beyond the scope of this work, we present and discuss some of the results in this section.

---

[5] These are results from an earlier version of the runtime system.

Figure 115 shows the northeast corner of the Andrews Forest (located in the upper-right corner of Figure 108), and the output breakthrough curve for each of the 17 streams therein. Each stream is uniquely numbered to match one of the graphs.

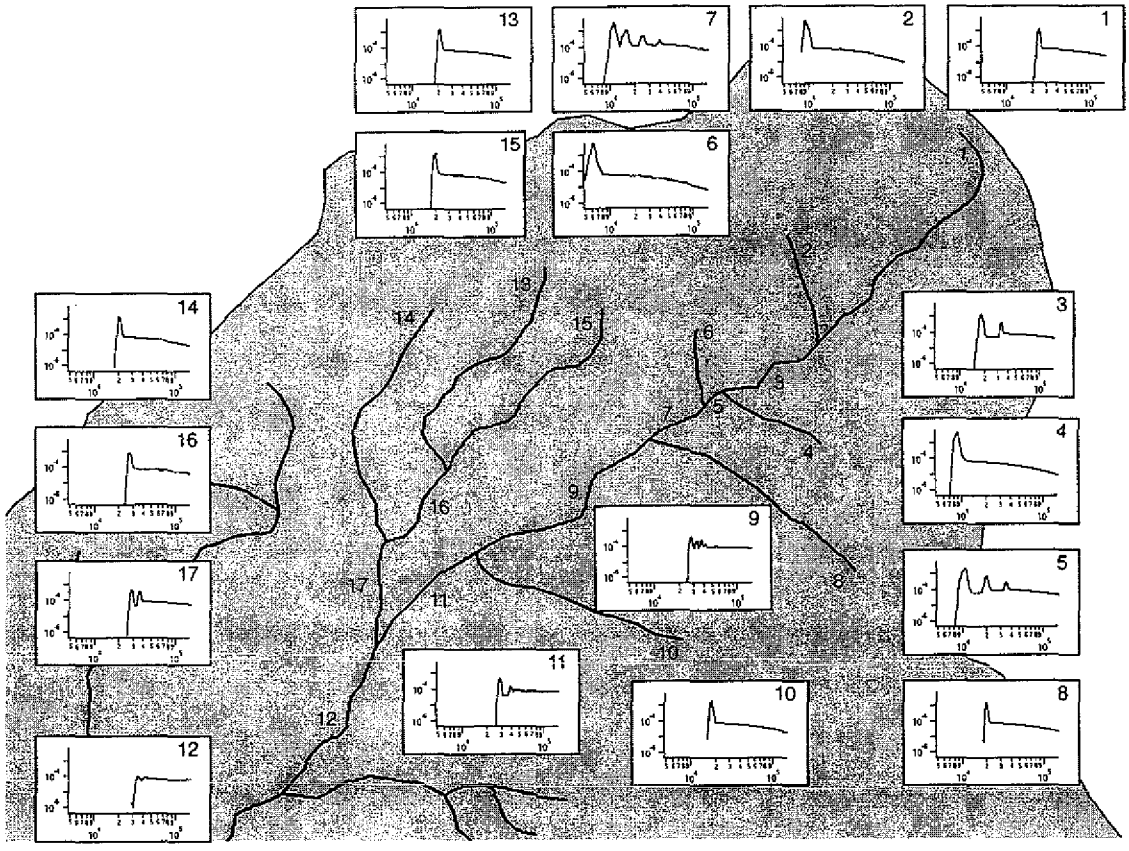

Figure 115. Breakthrough curves of different streams.

The shape of the breakthrough curves vary across the streams, from single-peak curves in the upstream-most segments, to different forms of multiple-peak curves in the inner streams. The effect that upstream segments have on downstream segments can be seen by inspecting steams 1, 2, and 3. The curves for streams 1 and 2 have a single peak, while the curve for stream 3 has two, indicating that the concentration outputs from streams 1 and 2 arrived at different times at stream 3. The curve for stream 7 has four distinct peaks, as influenced primarily by the upstream segments 6, 5, 4, and 3. The curve of the final stream segment,

12, is flatter than any other curve, and is compared with the curves of its connected upstream segments, 17 and 11, in Figure 116.

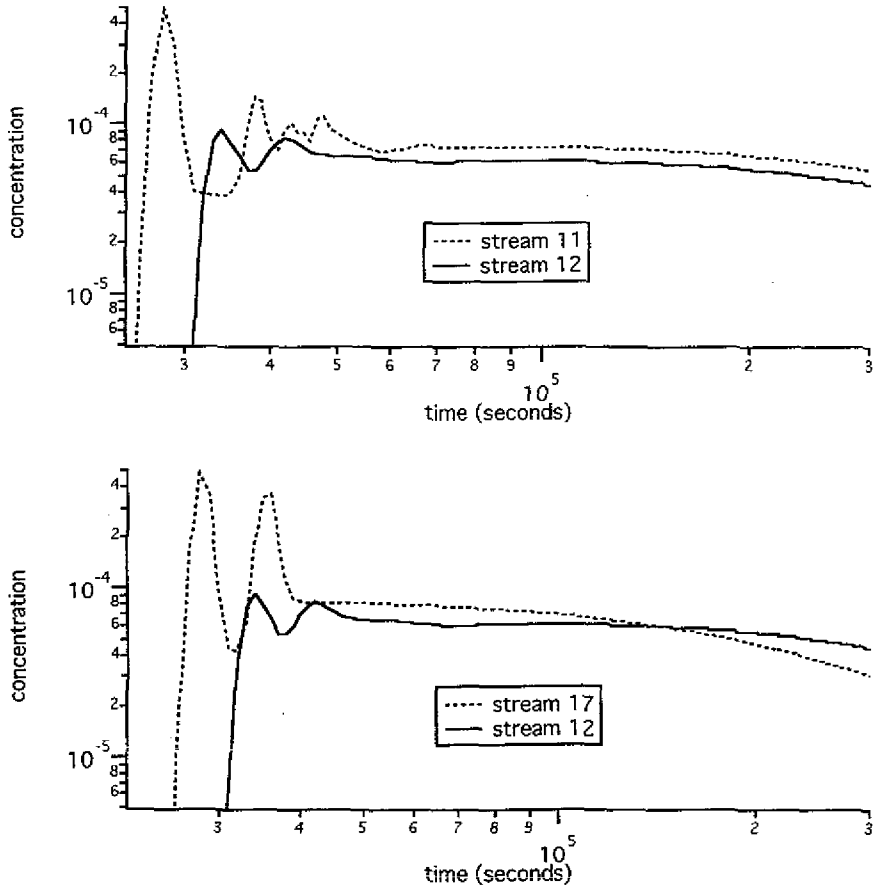

Figure 116. Comparison of the breakthrough curve of instance 12 with that of 11 and 17.

In both graphs in the figure, the solid line represents the breakthrough curve of stream 12 in Figure 115, and the dotted line represents the curve for one of its connected upstream segments. The breakthrough curves for streams 11 and 17 clearly influence the curve of stream 12. The first peak of streams 11 and 17 likely contribute to the first peak of stream 12, while the second peak of stream 17 and the smaller peaks of stream 11 contribute to the second peak of stream 12, which is somewhat flatter, likely due to the series of small peaks.

Once a model has been created, either from scratch or by coupling, it is often necessary to estimate the model parameters for a particular study site. This functionality is sometimes built into the model code itself, or provided by third-party software (Doherty 2004). Parameter estimation can be applied to coupled models just as it is to standalone models. Parameter estimation requires that a model be executed many times, each with slightly different parameters values. This can only practically be performed if the model can execute in an automated "batch" mode in which there is no user interaction, allowing the model to be executed many times, quickly. The implementation of PCICouple does not include a batch mode, but it could be added to support the optimization of coupled models.

The performance of the coupled model can be evaluated in terms of execution time and bandwidth used. The execution time, which was 4 minutes, 11 seconds on a 2.0 GHz AMD Athlon computer, is a combination of the execution times of each instance (which are the same if individually executed) and the communication time. The bandwidth used in the coupled model is 2.0 kb per send (500 data elements, one for each time step, multiplied by the size of the real data type, 4 bytes), for a total of 31.3 kb of bandwidth used in the 17-instance simulation.

Case Study Summary

This case study showed how the STAMMT-L model can be coupled to itself to achieve a more comprehensive simulation of the stream network of a watershed. We showed how the creation of value lists and data mappings can be automated through the use of the ArcMap GIS. In the next case study, we show how two models can be coupled to study the interaction between surfacewater runoff and groundwater.

Case Study: Simulating Runoff-Aquifer Interaction[6]

We describe in this study the initial results of a project we are conducting in collaboration with Alphonce Guzha at Utah State University, in which we are coupling the rainfall-runoff model TopModel to the groundwater-flow model MODFLOW to study the Tenmile Creek Watershed.

In year 2000, an initiative was founded called the WRIA 1 Watershed Management Project (WRIA1). This project seeks to address the increasing challenges of limited water supply, water quality degradation, and reduced numbers of Chinook salmon in Whatcom County, Washington State. Field research, data collection, monitoring, and computer modeling all play a role in the project and as a result there is a great deal of data available about the region making it an ideal area for study. In the central part of Whatcom county is the Tenmile Creek watershed. Figure 117 shows a photograph of Tenmile creek.

Figure 117. Tenmile creek.

It encompasses 65 miles of creeks and streams throughout 35 square miles (22, 670 acres), and includes Tenmile, Fourmile, and Deer creeks, as well as Crystal Springs, Barrett, Green and Fazon lakes.

We first motivate the purpose of the coupling, and then we describe how the incompatibilities between the models can be resolved and a coupling created using our approach. We then present the initial results of the coupled model.

---

[6] in collaboration with Alphonce Guzha, Utah State University

Motivation

It is typically the case that hydrological models either simulate surfacewater dynamics well and simulate groundwater dynamics poorly, or simulate groundwater dynamics well and simulate surfacewater dynamics poorly. Such models are good candidates for coupling, and one in particular is the rainfall-runoff model TopModel. TopModel (Beven 1997) simulates the amount of water that exits an area in response to rainfall. It is written in Fortran and consists of approximately 400 lines of model code. TopModel was not designed to accurately simulate groundwater and makes simplifying assumptions regarding it: the saturated zone is in equilibrium with a steady recharge rate over an upslope contributing area, and the water table is almost parallel to the surface such that the effective hydraulic gradient is equal to the local surface slope. A more accurate simulation can be achieved by incorporating the simulation of the groundwater performed by ModFlow into the simulation of surfacewater-runoff performed by TopModel in a similar way as Example 1 of Chapter 5 in which the SWMM model was coupled ModFlow. ModFlow, introduced in detail in Example 1 of Chapter 5, simulates the movement of groundwater in the saturated zone.

Coupled Model Design

In this coupled model, the water table head value that is calculated by TopModel is replaced with the more accurate head value simulated by ModFlow. The *hnew* variable in ModFlow represents the water table head. It is an array in which each element represents the head in a different grid cell. The *sd* variable in TopModel represents the water table head beneath a subcatchment, measured as a depth below the land surface, in meters. There is a loop in TopModel that iterates through each subcatchment, so this scalar represents the head beneath a different subcatchment on each iteration of that loop. In our study site in the Tenmile Creek Watershed, there are 3 subcatchments. These are shown in Figure 118 as they appear in the ArcMap GIS, including the stream network.

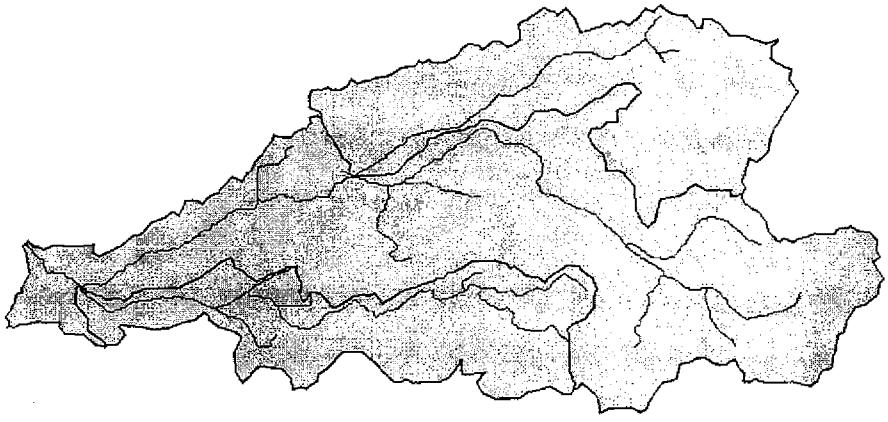

Figure 118. Three subcatchments in the Tenmile Creek Watershed.

In this initial work we divided the watershed into four grid cells that are simulated by ModFlow and we expect to increase the resolution of the grid substantially in our future work. Figure 119 shows the three subcatchments superimposed on the four grid cells.

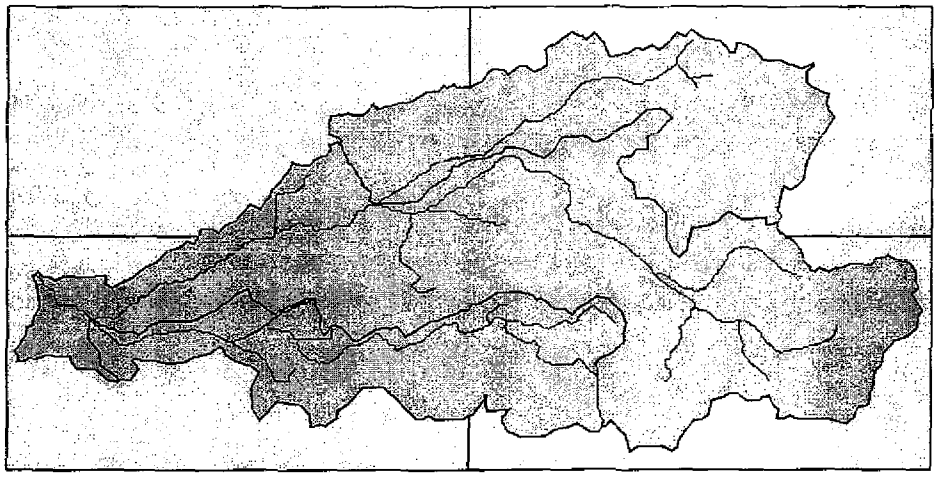

Figure 119. Subcatchments superimposed on a grid.

This difference in the spatial distribution of the modeled quantities is common and is resolved in the CDL through data mappings. Before explaining how this spatial incompatibility was resolved, we present the coupling description.

The Coupling Description

The coupling description is shown in Figure 120. As in most couplings, the models communicate within their time step loops.

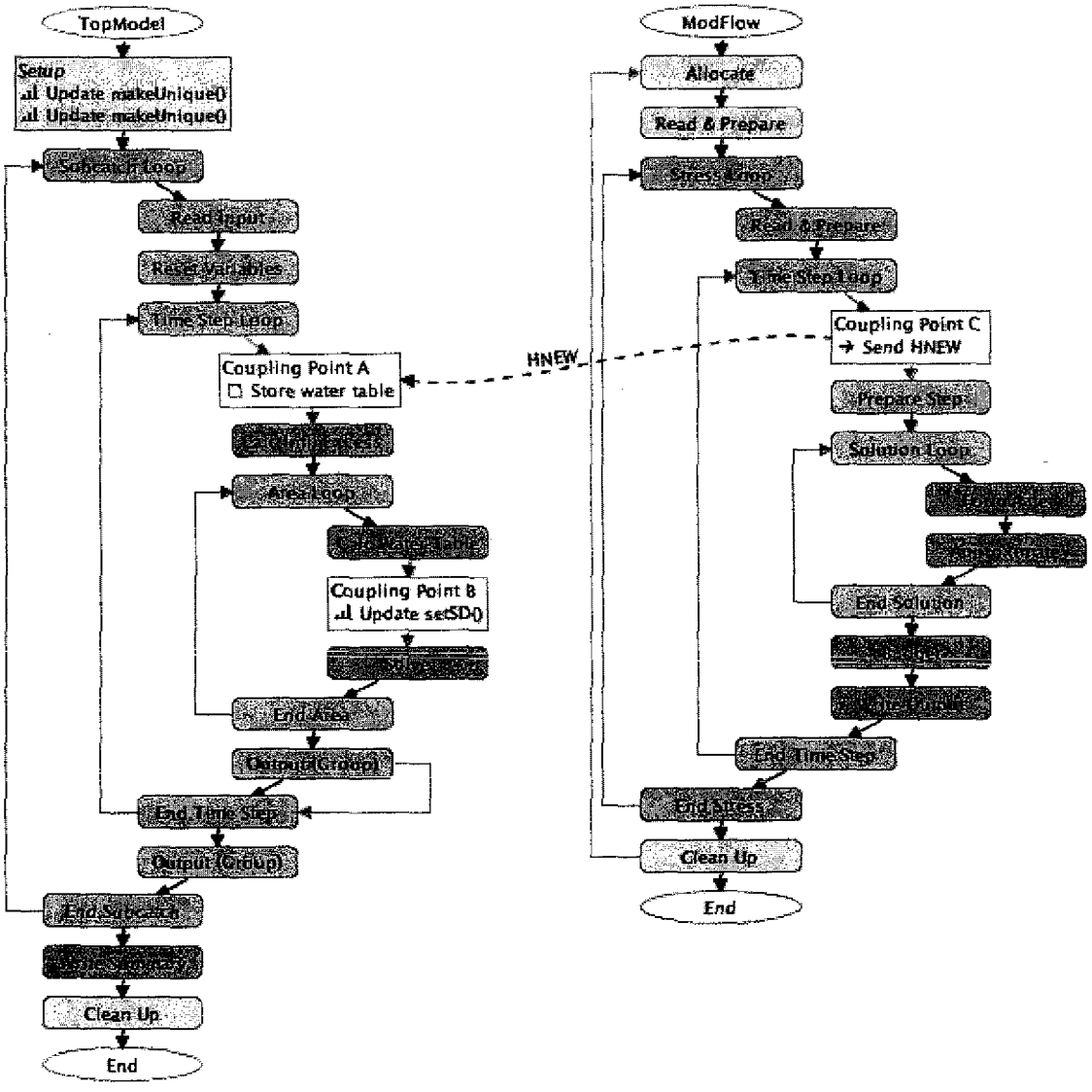

Figure 120. The coupling description.

As long as the models start at the same point in simulation time, and use the same time step length, then the models will remain coordinated in simulation time. ModFlow though, usually uses a long time step length, on the order of days

or weeks due to the slow speed at which groundwater moves, and TopModel can use either a short step length to study individual storm events (on the order of hours), or a longer step length to study long term trends (on the order of days). Differing step lengths can be accommodated by adjusting the activation frequencies of actions. For example, if ModFlow uses a step length of 2 days, and TopModel uses a step length of 1 day, then the greatest frequency at which the models can communicate is every 2 days, and the activation frequencies of the actions in TopModel would be set to 2, and set to 1 in ModFlow. In this study the models are parameterized to use a common step length of 1 day and we expect to use differing step lengths in each model in our future work.

Inspection of the model PCIs reveals that TopModel simulates each subcatchment individually, each in a different iteration of the subcatchment loop. The time step loop is therefore executed in its entirety for each subcatchment, in our case, three times. The time step loop in ModFlow though, is executed only once (the outer Stress loop is used to coordinate the time step loop, but each time step is only executed once). During execution of the coupled model, the models would remain coordinated in simulation time throughout their time step loops, but after the time step loops finish, ModFlow would exit, and TopModel would begin its simulation of the next subcatchment, during which it would expect to communicate with ModFlow, resulting in an error. This structural incompatibility can be resolved by using three instances of TopModel, where each instance simulates a different subcatchment. Since each instance simulates only a single subcatchment, the subcatchment loop is executed only once in each instance, resulting in the time step loop being executed only once in each instance, just as in ModFlow.

The coupling description includes four action lists, one at each of the three expanded coupling points (A, B, and C), and one at TopModel's "Setup" block. We explain the purpose of each.

Setup Block: There are three instances of TopModel in this coupling that execute concurrently, each of which should simulate a different subcatchment. When executed though, each instance will read the same input files, resulting in all the instances simulating a subcatchment with the same characteristics. In order for each instance to simulate a different subcatchment, each instance must use a different input characteristics file. The variables that store the filenames used by TopModel are accessible at the Setup block and are the *subcats* and *outputs* variables. These variables store the filenames of the input characteristics file and the output file, respectively. Two Update Actions are added to this block, each of which applies the custom update function, *makeUnique*, to one of these variables, making them unique (the output filename must be unique since multiple processes cannot write to the same file concurrently). The function simply prepends the instance identifier (accessible in all update functions via the instanceID variable) to the filenames, making them unique.

Coupling Point A: Although the value of the *hnew* variable sent from ModFlow needs to be used at Coupling Point C, the value is sent to Coupling Point A and stored because Coupling Point C is located within a loop, and communicating with ModFlow at that point would cause the models to become unsynchronized, similar to the subcatchment loop incompatibility discussed earlier.

Coupling Point B: Since TopModel needs to use ModFlow's *hnew* variable, a Send Action is added to Coupling Point B in ModFlow, which sends the variable's value to TopModel, making it accessible at Coupling Point A.

Coupling Point C: To set the value of TopModel's *sd* variable, an Update Action is added to Coupling Point A which applies the custom update function, *setHead*, which sets the value of the *sd* variable based on the value of ModFlow's *hnew* variable. Note that ModFlow's *hnew* value is an elevation, whereas TopModel's *sd* variable is a depth. In order to set the *sd* value to the *hnew* value,

the elevation must be converted into a depth. This requires knowledge of the elevation of the surface, since the depth is equal to the difference between the surface elevation and the water table elevation. In this function we assume a constant surface elevation (10 feet). These custom update functions were written in Fortran, and account for any differences in units between the variables upon which they operate. Here, the depth value is calculated in feet and converted into meters (my multiplying by 3.28) when the value of *sd* is set. The source code for the *setHead* function is shown in Figure 121.

```
subroutine setHead(instanceID,sd,head)
    integer          instanceID
    real             sd(30)
    double precision head

    sd = (10.0 - head) / 3.28
end
```

Figure 121. Source code for the setHead function.

Differences in units are easy to resolve, but differences in spatial distribution are not so straightforward. Although *hnew* is an array representative of the groundwater height across a regular grid, the value received by TopModel must be a scalar that represents the groundwater height below the subcatchment being simulated. This transformation is accomplished via the data mapping shown in Figure 122, and is assigned to the Send Action in the coupling description. The data mapping indicates that there are three instances of TopModel and one instance of ModFlow, and it describes how the groundwater height values from each grid cell are weighted and combined to arrive at a value representative of the groundwater height below each subcatchment. Notice that since *hnew* is an array, the data mapping describes how each element is communicated and transformed. For example, the value of *hnew* received by instance 1 of TopModel is a combination of the groundwater heights of the cells below it, cells 1, 2 and 4 (elements 1, 2 and 4 of the *hnew* array), shown in Figure 122.

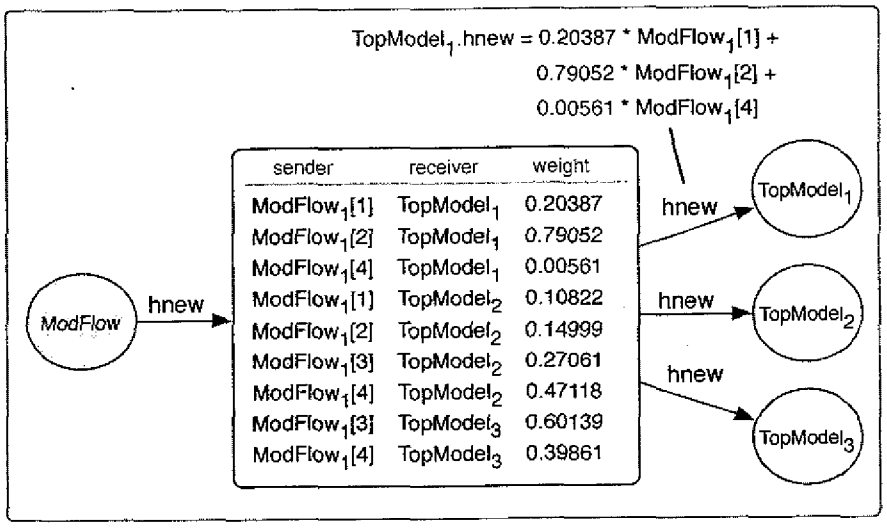

Figure 122. The data mapping relates the regular grid to the irregular subcatchments.

Specifically, the value is composed of 20.4% of the value from cell 1, 79.1% of the value from cell 2, and 0.5% of the value of cell 4.

This data mapping was created automatically via a script that we wrote in ArcMap. The grid cell polygons within ArcMap were numbered according to which element of *hnew* they are associated with, and the subcatchment polygons were numbered according to which instance of TopModel simulates it, as shown in Figure 123.

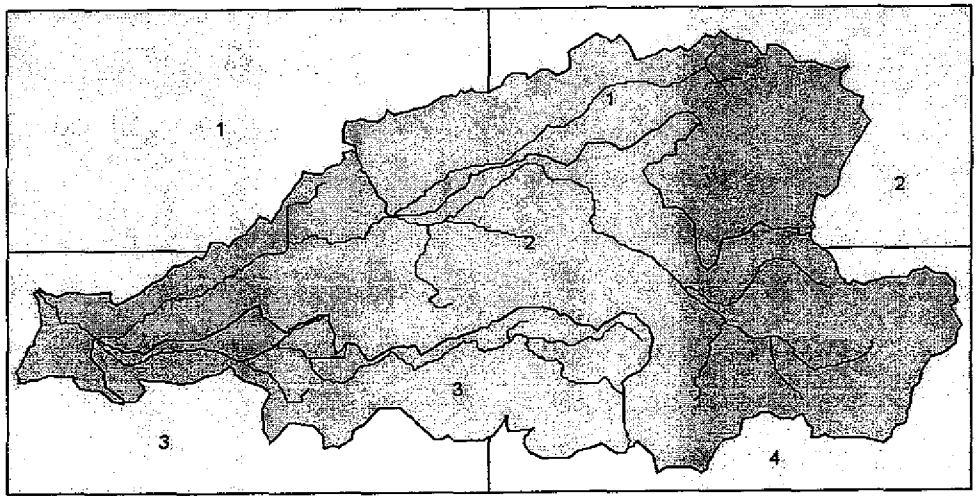

Figure 123. Spatial units are numbered by array element or instance id.

The script then performs a common GIS operation called an overlay. The script determines which polygons overlap with each other, and by how much, and then generates an output file in the proper data mapping format. In this way, the precise relationship between the variables can be established. Note though, that data mappings are general-purpose and can describe the relationship between any kind of data, not just spatial data. In this case we use a GIS to create the mapping, but other third-parties could be used to create mappings between other kinds of data. The script that we wrote is shown in Figure 124.

```
Public Sub createDataMapping()
  Dim pMxDoc As IMxDocument, pMap As IMap, pFilter As IQueryFilter
  Dim pIntersectLayer As IFeatureLayer, pOverlayLayer As IFeatureLayer
  Dim pOverlayFCursor As IFeatureCursor, pIntersectFCursor As IFeatureCursor
  Dim pIntersectTopo As ITopologicalOperator As Double, theProportion As Double
  Dim pIntersectFeature As IFeature, pOverlayFeature As IFeature
  Dim pSpatialFilter As ISpatialFilter, pOverlayArea As IArea, newArea As IArea
  Dim pIntersectFClass,pOverlayFClass As IFeatureClass
  Set pMxDoc = Application.Document
  Set pMap = pMxDoc.FocusMap
  Set pActiveView = pMap

  If Not TypeOf pMap.Layer(0) Is IFeatureLayer Then Exit Sub
  If Not TypeOf pMap.Layer(1) Is IFeatureLayer Then Exit Sub
  Set pIntersectLayer = pMap.Layer(0)
  Set pOverlayLayer = pMap.Layer(1)

  Open "data_map.txt" For Output As #1
  headerLine = Format(Now, "mm/dd/yyyy")
  Print #1, "Created by ArcMap on " & headerLine

  Set pIntersectFClass = pIntersectLayer.FeatureClass
  Set pOverlayFClass = pOverlayLayer.FeatureClass
  Set pFilter = New QueryFilter
  pFilter.WhereClause = ""
  Set pIntersectFCursor = pIntersectLayer.Search(pFilter, False)
  Set pIntersectFeature = pIntersectFCursor.NextFeature

  While Not pIntersectFeature Is Nothing
    Set pIntersectTopo = pIntersectFeature.Shape
    Set pSpatialFilter = New SpatialFilter
    pSpatialFilter.GeometryField = pIntersectFClass.shapeFieldName
    Set pSpatialFilter.Geometry = pIntersectFeature.Shape
    pSpatialFilter.SpatialRel = esriSpatialRelIntersects
    Set pOverlayFCursor = pOverlayFClass.Search(pSpatialFilter, False)
```

Figure 124. The script that generates the data mapping.

```
    Set pOverlayFeature = pOverlayFCursor.NextFeature
    While Not pOverlayFeature Is Nothing
      Set pOverlayArea = pOverlayFeature.Shape
      Set newGeometry = pIntersectTopo.Intersect(pOverlayFeature.Shape,
                        pIntersectFeature.Shape.Dimension)

      Set newArea = newGeometry
      theProportion = newArea.Area / pOverlayArea.Area
      iModel = pIntersectFeature.Value(pIntersectFeature.Fields.FindField("Model"))
      iCoupleID =
           pIntersectFeature.Value(pIntersectFeature.Fields.FindField("CoupleID"))
      oModel = pOverlayFeature.Value(pOverlayFeature.Fields.FindField("Model"))
      oCoupleID = pOverlayFeature.Value(pOverlayFeature.Fields.FindField("CoupleID"))

      Print #1, iModel & ":" & iCoupleID & " " & oModel & ":" & oCoupleID & " " &
              theProportion
      Set pOverlayFeature = pOverlayFCursor.NextFeature
    Wend
    Set pIntersectFeature = pIntersectFCursor.NextFeature
  Wend
  Close #1
End Sub
```

Figure 124. The script that generates the data mapping (cont'd).

Once the model input files have been prepared, the coupling can be executed via PCICouple. The runtime environment provided by PCICouple is shown in Figure 125.

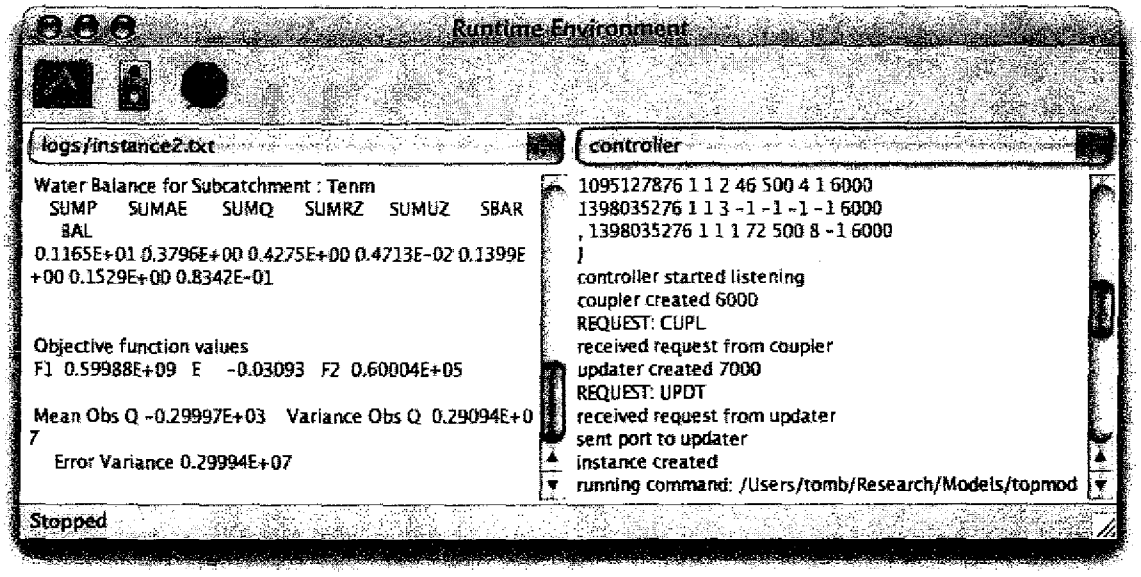

Figure 125. The runtime environment provided by PCICouple.

The runtime system will start one instance of ModFlow and three instances of TopModel, along with any necessary couplers and updaters. The instances will communicate with each other throughout their simulations, and each instance will write its output files which can then be analyzed. The output from any process can be viewed by selecting it from one of the two process menus. In the figure, the output from instance 2 is shown on the left (from TopModel in this case), and the output from the controller is shown on the right. The buttons in the toolbar are (from left to right): *edit coupling description, execute coupled model,* and *stop execution.*

Results

We are interested in investigating how variations in the water table head simulated by ModFlow affect the catchment outflow simulated by TopModel. We performed two executions of the coupled model in which ModFlow was parameterized to simulate different water table heads, and we compared the TopModel output from these runs to the output simulated by TopModel in an uncoupled simulation, which serves as a control.

We created the input file sets for the models for each of the three cases using initial data that we collected for the Tenmile Watershed. The precipitation input is shown in Figure 126.

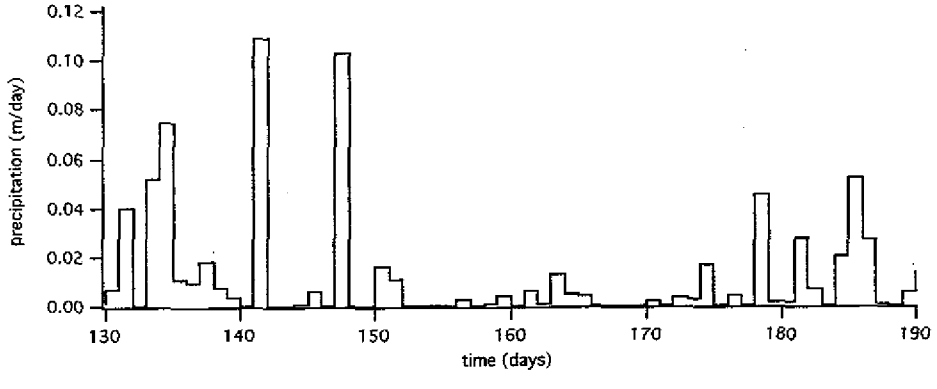

Figure 126. The precipitation input used by TopModel in all three cases.

The overland flow from saturated areas simulated by TopModel in each case is shown in Figure 127 as a function of time.

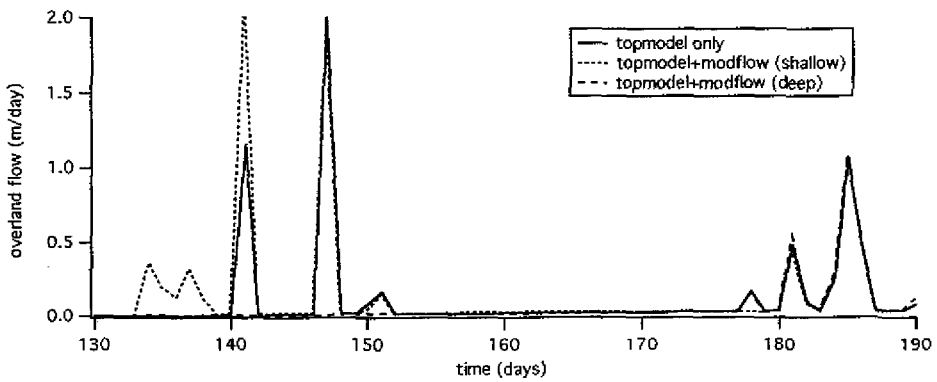

Figure 127. Comparison of the overland flow simulated by TopModel in each case.

The output of TopModel when it was not coupled to ModFlow is indicated by the solid line (labeled "topmodel only" in the legend), and the output from TopModel in each of the two coupled model runs is indicated by the dotted and dashed lines. The dotted line (labeled "topmodel+modflow (shallow)" in the figure) represents the output from TopModel when it was coupled to ModFlow, where ModFlow was parameterized to simulate a water table head that is half the depth as simulated by TopModel alone (the control case). The dashed line (labeled "topmodel+modflow (deep)" in the figure) represents the output from TopModel when it was coupled to ModFlow, where ModFlow was parameterized to simulate a water table head that is twice the depth as simulated by TopModel alone (the control case).

The simulated outflow in each of the coupled cases is consistent with our expectations: in the case of the shallow water table, the outflow is greater than the control case, indicating that baseflow from the saturated zone is contributing to the outflow. In the case of the deep water table, the outflow is less than the control case, indicating that the saturated zone contributes little to the outflow.

The recharge to the saturated zone from the unsaturated zone as simulated by TopModel in each of the three cases is shown in Figure 128. The recharge is shown as a function of time.

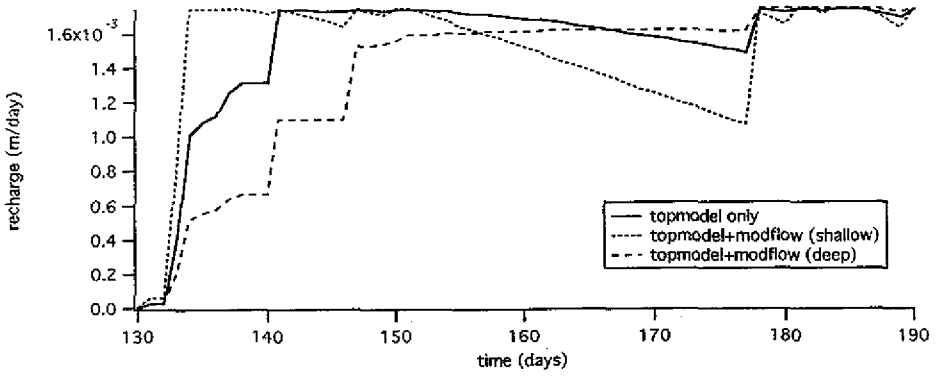

Figure 128. Comparison of the recharge simulated by TopModel in each case.

Again, the output from TopModel is consistent with our expectations: in the shallow water table case, there is considerable recharge to the saturated zone indicating that the unsaturated zone is at capacity and excess runoff is likely. In the deep water table case, there is initially little recharge to the saturated zone, indicating that the unsaturated zone is absorbing most of the rainfall and it is not until day 178 that the unsaturated zone is close to capacity and contributes to the saturated zone.

The coupled model runs had an execution time of approximately 54 seconds on a 1.5 GHz PowerPC G4 computer, 8 seconds of which was startup time (14.8%). The coupler sent a total of 1.0 MB of data and received 0.81 MB.

Case Study Summary

This case study showed how the rainfall-runoff model TopModel can be coupled to the groundwater-flow model ModFlow to achieve a more accurate simulation. We showed how complex spatial relationships can exist between the variables of different models, and how data mappings, created in an automated way via a GIS, can describe these relationships.

Summary

This chapter presented three case studies that demonstrate how the CDL is used to create coupled models in scientific studies. In the first case study we compared an existing coupling between ModFlow and DAFlow to a re-creation of that coupling using our approach. The second case study showed how many instances of a lumped model, STAMMT-L, could be coupled together to enable spatially-distributed simulation across an entire watershed. The third study showed how two models can be coupled together to study the interaction between rainfall-runoff and groundwater by coupling together TopModel and ModFlow. The next chapter presents the conclusions of this work.

CHAPTER VIII

CONCLUSIONS

The process of model coupling allows existing models to be combined such that they affect each other's simulations as they execute. This process is intimately tied to the source codes of the models. Existing techniques require scientists to work directly with these model codes, making the process prohibitively difficult. In this dissertation, we address the problem by allowing scientists to work at a higher level of abstraction through the use of model interfaces for coupling.

Specifically, this dissertation contributes:

- *the design of a representation for model coupling interfaces, called the Potential Coupling Interface.*

We introduced the concept of coupling potential, and showed how it can be conveyed concisely through a visual interface. This reusable interface is easy to create with our software assistants, PCICreate and PCICouple. The coupling potential of a model can be quickly understood by inspecting a PCI, saving scientists a great deal of (often redundant) effort. Since the PCI is derived from model codes, we showed how it is capable of automatically instrumenting the model codes, a necessary step in creating a coupled model that is difficult and error-prone in existing approaches.

The second contribution of this work is

- *the design of a language for describing the behavior of coupled models, called the Coupling Description Language.*

The primary purpose of the PCI is to serve as the basis for describing coupled models. One of the most challenging tasks in creating a coupled model is identifying and resolving incompatibilities between models, and reasoning about how the models should influence each other. Through a series of hypothetical examples and real case studies, we showed how the PCI reveals incompatibilities between models, and how the coupling language provides a means for resolving them. The distributed, visual coupling language imposes no restrictions on the kinds of models coupled, or the ways in which the models interact. Arbitrarily complex interactions between many different kinds of models are supported. The InCouple approach to model coupling is based on model PCIs, so the ability to create a PCI from a model code is a prerequisite for coupling models. There may be cases where it is not possible to create a PCI that accurately describes the control structure of a model code (e.g. Time Warp simulations).

The third contribution of this dissertation is

- *the evaluation of a prototype coupling environment for hydrological models.*

Through the implementation of a proof-of-concept coupling environment, we demonstrated the practicality of our approach to model coupling. Although we populated the coupling environment with hydrology models, the environment itself does not contain any customizations that are specific to hydrology models, making it suitable for model coupling in many domains. The framework is open source and available for download and use by scientists.

Here we discuss how our work compares to and complements existing model coupling frameworks. Since our approach is a variant of the communica-

tion approach described in Chapter 2, we will limit the comparison to these existing approaches. We compare the approaches in terms of the coupling process and how it is supported by our approach and existing approaches.

With respect to finding models to couple, existing work offers little support for this task. Locating models on the internet is difficult because there are so many places where models are available, and the information online is generally sparse, requiring the scientist to download and evaluate models individually. Even once emerging model metadata standards are widely accepted and applied, these do not indicate to the scientist the coupling-compatibility between models. The PCI though, captures just this information and can be incorporated into these standards to facilitate locating and comparing models in terms of their coupling compatibility, greatly simplifying the task of finding models to couple.

With respect to understanding a model's state variables and accessible locations throughout a model code, existing approaches offer little or no assistance, whereas in our approach, we focus specifically on the knowledge required to create a coupled model. These assume that the scientist is an expert in the model and its source code, despite the severe challenge required to understand them. The PCI captures this information and clearly presents it to the scientist. Documenting models, coupled and otherwise, is essential to their proper use by other scientists. In our approach, the operational coupling description serves directly as documentation for the coupling.

Resolving incompatibilities between models and instrumenting the model codes are central tasks of model coupling and are supported in both existing approaches and our approach. Extensive existing work has been done in the area of spatial regridding and efficient communication of large datasets. Recent work has focused on network applications of this in computational grids. Since several of these frameworks offer more powerful transformations and efficient communi-

cation, our work could be adapted such that the model codes are automatically instrumented with calls to one of these frameworks or libraries. Since the instrumentation in our models is generic, scientists can capitalize on the work of others by reusing PCIs in different couplings whereas in existing approaches, instrumentation is typically coupling-specific.

The concepts and techniques presented in this dissertation describe a novel way of reasoning about and constructing coupled models. The use of model interfaces for coupling provides an unprecedented level of ease and flexibility in creating coupled models, significantly broadening the practicality of the practice of model coupling in the scientific community.

APPENDICES

Appendix A: PCI Collection

This appendix presents the Potential Coupling Interfaces (PCI) for each of the hydrological models referenced throughout this work. The creation of the PCI for a model requires a thorough understanding of the model, both in its use and its implementation as a computer program. Although we have worked with each of these models as part of our research, we may not have achieved a complete understanding of them, and hence it is possible that more accurate and informative PCIs for these models could be created by scientists who possess a greater level of understanding of the models. Nonetheless, we present the PCIs for each model based on our knowledge. There is no single "correct" PCI for a model, and different PCIs for a model could be equally informative and accurate, so these represent only one possible PCI for each model. References for these models is given in Table 4 in Chapter 4.

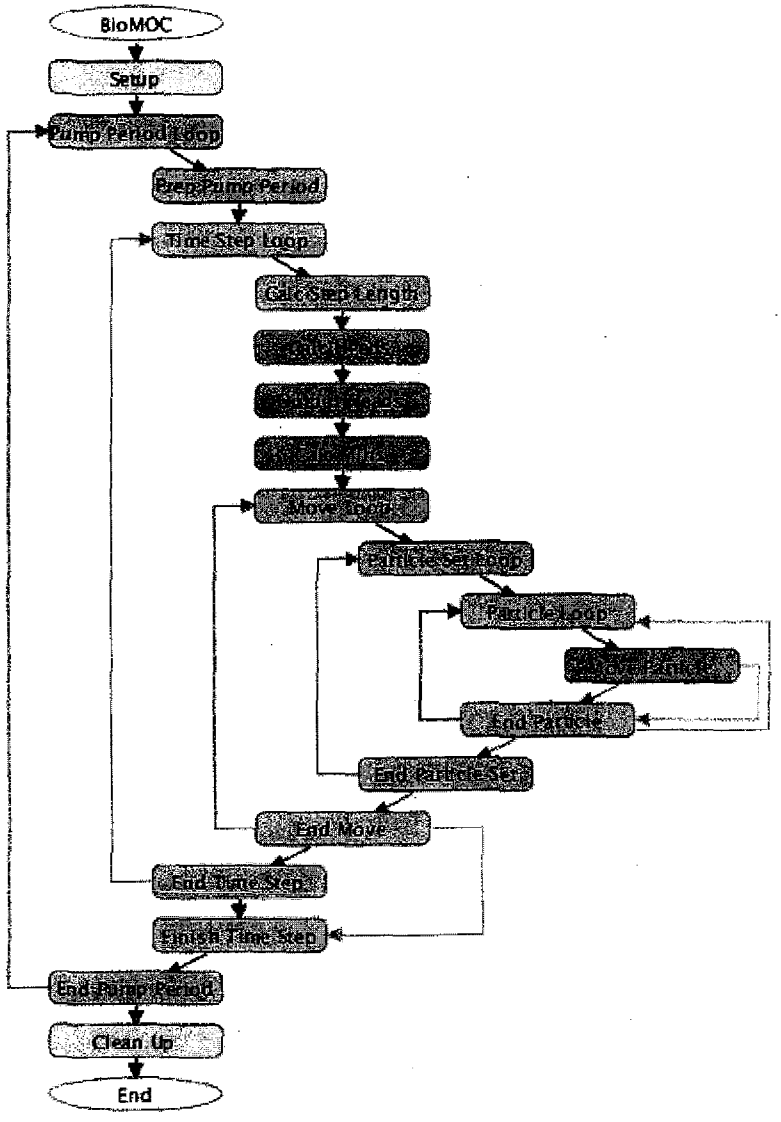

Figure 129. A PCI for BioMOC.

Figure 129 shows a PCI for the groundwater-transport model BioMOC. Notice the nested temporal loops and the block that sets the time step length. Couplings involving this model must explicitly coordinate the points in simulation time at which communication takes place.

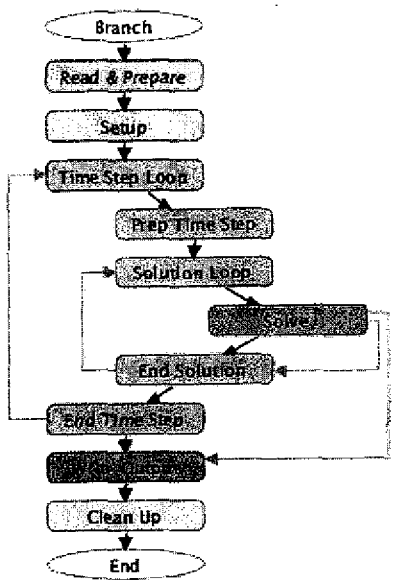

Figure 130. A PCI for Branch.

A PCI for Branch is shown in Figure 130, and a PCI for FourPt is shown in Figure 131. These models simulate the movement of water through a network of inter-connected channels, similar to DAFlow.

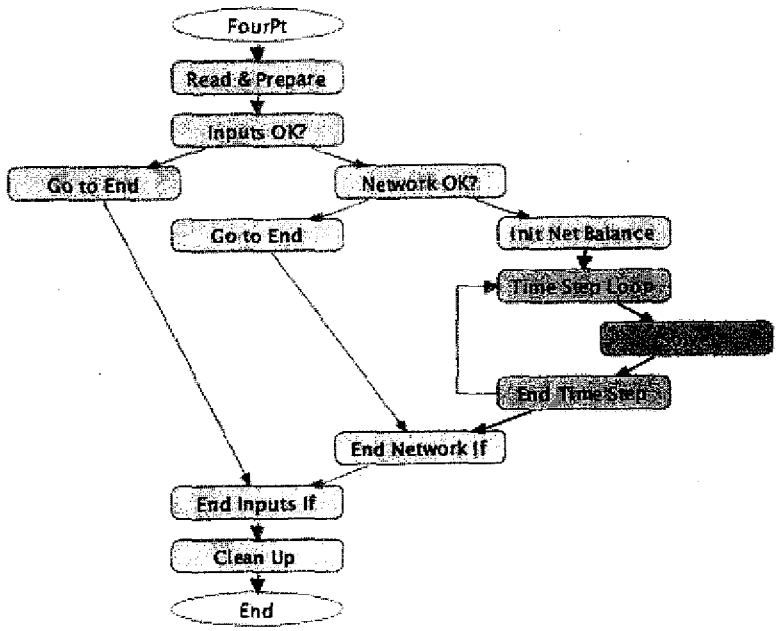

Figure 131. A PCI for FourPt.

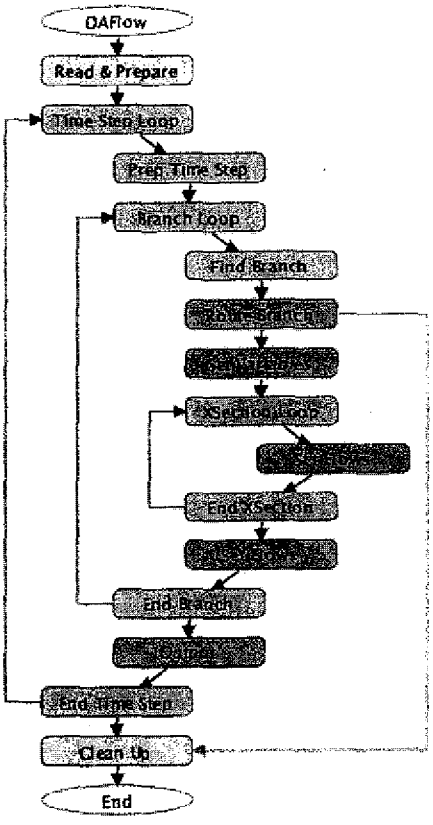

Figure 132. A PCI for DAFlow.

The PCI for DAFlow used throughout this text is shown in Figure 132. Notice how there are two spatial loops nested within the temporal loop: one that iterates over the branches in the network, and one that iterates over the cross sections within each branch.

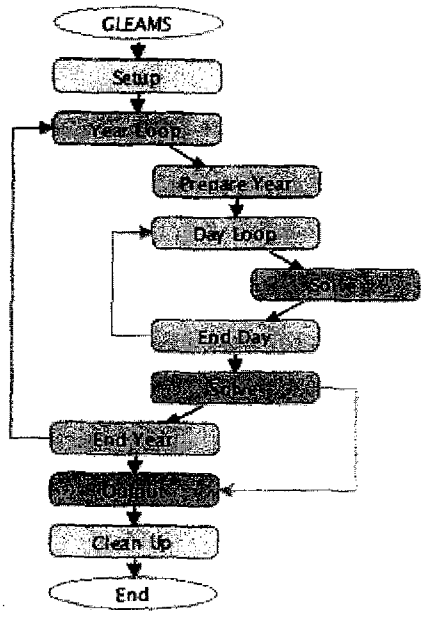

Figure 133. A PCI for GLEAMS.

The PCI used for GLEAMS in the text is shown in Figure 133, and the PCI for a similar model SHAW is shown in Figure 134.

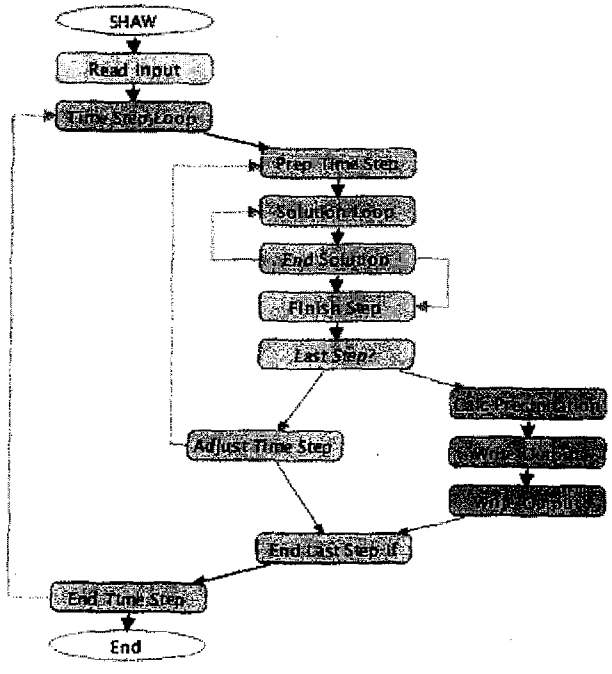

Figure 134. A PCI for SHAW.

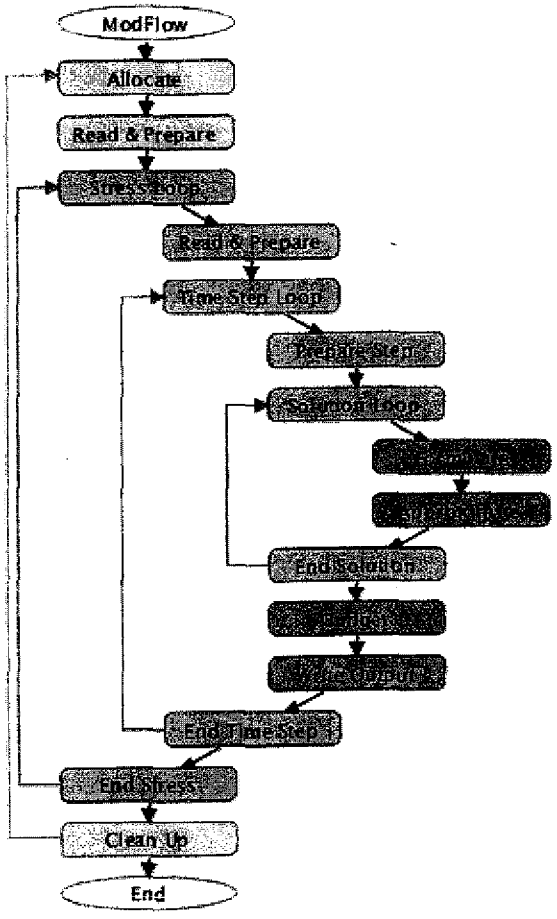

Figure 135. A PCI for ModFlow.

The PCI for ModFlow used throughout this text is shown in Figure 135. Notice the nested temporal loops and the lack of a spatial loop.

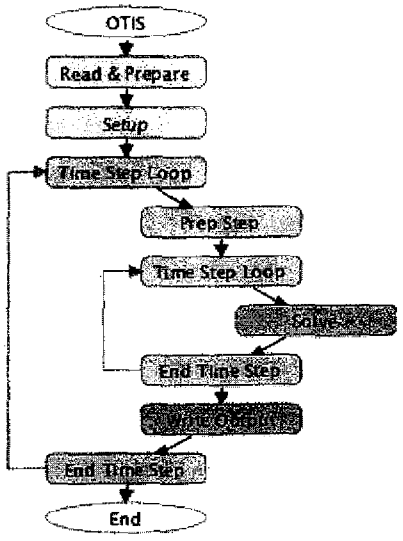

Figure 136. A PCI for OTIS.

A PCI for the OTIS model is shown in Figure 136 and the PCI used for the STAMMT-L model is shown in Figure 137. Both models simulate surfacewater transport. Notice how STAMMT-L does not possess any loops and hence can only communicate with other models before and after its computation.

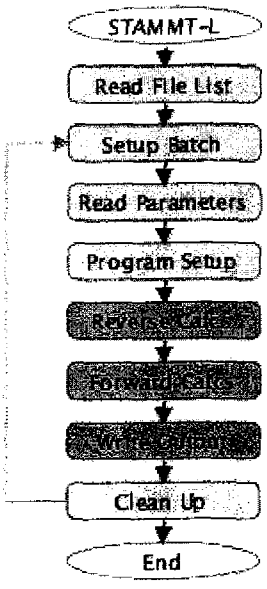

Figure 137. A PCI for STAMMT-L.

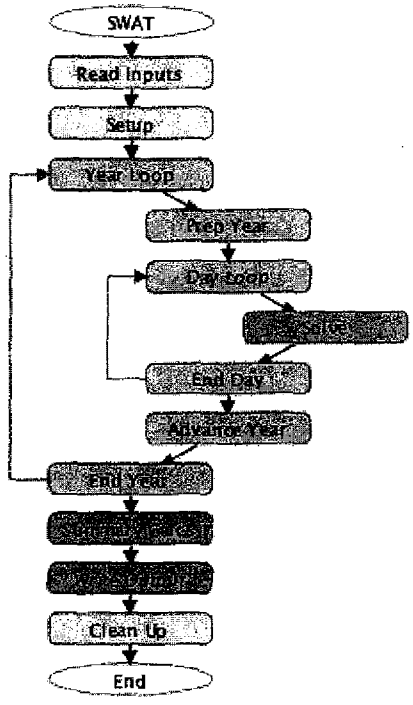

Figure 138. A PCI for SWAT.

A PCI for the rainfall-runoff model SWAT is shown in Figure 138.

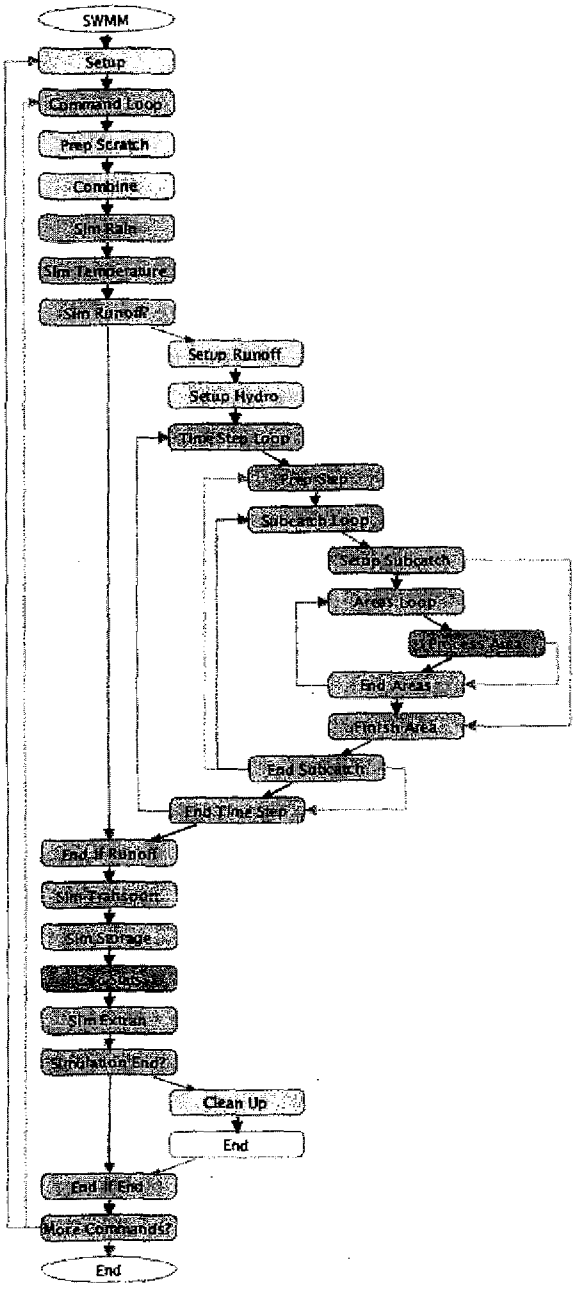

Figure 139. A PCI for SWMM.

The PCI for SWMM used throughout this text is shown in Figure 139. This PCI only includes coupling points within the runoff computation, but a complete PCI would include points within the other computations as well (transport, storage, etc.)

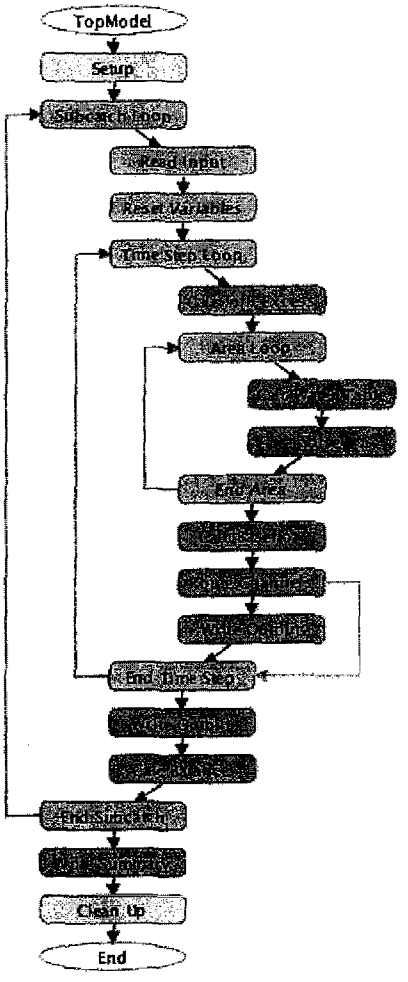

Figure 140. A PCI for TopModel.

The PCI used for TopModel throughout this text is shown in Figure 140.

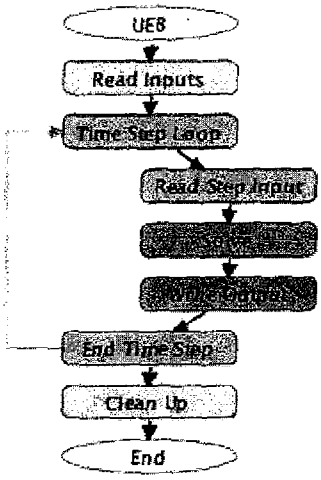

Figure 141. A PCI for UEB.

A PCI used for UEB throughout this text is shown in Figure 141.

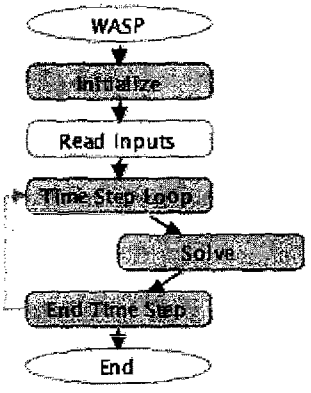

Figure 142. A PCI for WASP.

A PCI for the receiving water model WASP is shown in Figure 142.

Appendix B: Guidelines for Annotating Model Codes

This section describes a series of guidelines regarding how model codes should be annotated. The most important consideration when annotating a model code is that it is done in a way which describes the model code's full coupling potential. One of the key benefits of the InCouple approach to model coupling is that the interfaces are reusable: created only once, the interface can be reused in any future coupling endeavor. This requires that the model codes be annotated such that all the variables that could potentially be involved in any future coupling are anticipated and annotated. It is important that any variable that possesses any domain-level significance be annotated. In addition to the state variables themselves, auxiliary variables associated with them should be annotated as well. For example, we saw in the text how each element of ModFlow's *hnew* array represents the groundwater height at a different grid cell. Not only should the *hnew* array be annotated, but so too the variables that describe how many cells there are: *nrow*, *ncol*, *nlay*.

Certain loops in a model code that have domain-level significance, such as time loops or space loops, are good places for annotations. Annotations are typically placed both before and after the loop of interest, and within the loop, at its start and end. It is important though, to pay attention to the kind of loop, and its iteration frequency. Placing annotations within a tight loop may have serious performance impacts. Equally expressive annotations may be possible that do not incur the performance penalty. For example, suppose a loop iterates over an array, setting the value of each element. If an annotation was placed outside the loop rather than inside of it, the full array could be accessible after, rather than each of its elements within the loop.

Since the PCI is used to convey the overall structure of a model code to the scientist, it is important to place annotations between the major phases of the

model code, such as before and after the initialization phase, and before and after results are output. Making sure that the prominent phases of a model code are well represented will dramatically improve the readability of the PCI.

Appendix C: Adding Custom Data Mappings

This appendix describes how data mapping input files are added to a coupling description within PCICouple. Once the scientist has created a data mapping input file, it must be added to PCICouple before it appears as an available data mapping within Send Actions. The first step is to click the data mapping button in the inspector window, which shows the list of available data mappings, and it initially contains only the built-in mappings as shown in Figure 143 (left). To add a data mapping to the list, the scientist clicks the add button (marked by a plus symbol) and selects the data mapping file.

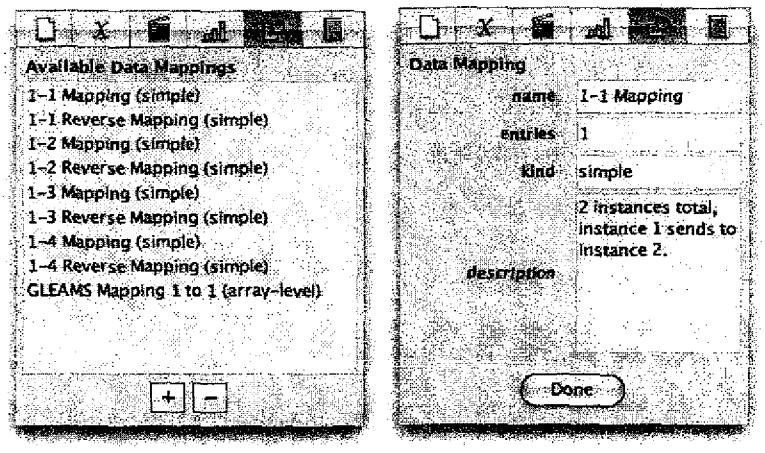

Figure 143. Available data mappings (left) and the mapping details (right).

When a data mapping is in the list is double-clicked, the data mapping details are shown in the inspector, as shown in Figure 143 (right).

Appendix D. Adding Custom Update Functions

Although the coupling environment includes a set of general purpose update functions such as *set*, *sum*, and *average*, the way in which a value from one model affects the value of another is complex, and often requires non-trivial calculations. For this reason, PCICouple allows scientists to add their own update functions to coupling descriptions. This process involves two steps: first, the scientist must write the function and compile it into the updater program, then second, the scientist must register the function with PCICouple. Each step is described in turn.

The first step in using a custom update function in a coupling description is to write the function source code, in whatever language the scientist prefers (Fortran in our implementation). This function is then added to the source code of the updater program, provided with PCICouple. The function source code is simply added to the source code of the updater, so that couplers can invoke it. Currently the scientist must add a wrapper function around each custom update function that invokes send and receive library calls (using our custom API) to their functions which retrieve the function arguments from the coupler and return them. Figure 144 shows the source code of the wrapper function for the *setHead* update function.

```
subroutine setHeadWrapper(instanceID,socket,sd,head)
    integer            socket,instanceID,sd_len,head_len
    real               sd(30)
    double precision   head(2,2)

    call receive( sd, 30, 4, socket, sd_len )
    call receive( head, 4, 8, socket, head_len )

    call setHead( instanceID, sd, head )

    call send( sd, sd_len, 4, socket )
    call send( head, head_len, 8, socket )
end
```

Figure 144. The communication wrapper function for the setHead function.

This is an implementation simplification and it could easily be changed such that the arguments are automatically sent and received so that the scientist must only add their functions to the updater source code without writing a wrapper function.

The second step in using a custom update function is to tell PCICouple about the function. This is accomplished in PCICouple by clicking the update functions button in the inspector window to view the list of available update functions. The built-in functions will appear in the list, as shown in Figure 145 (left).

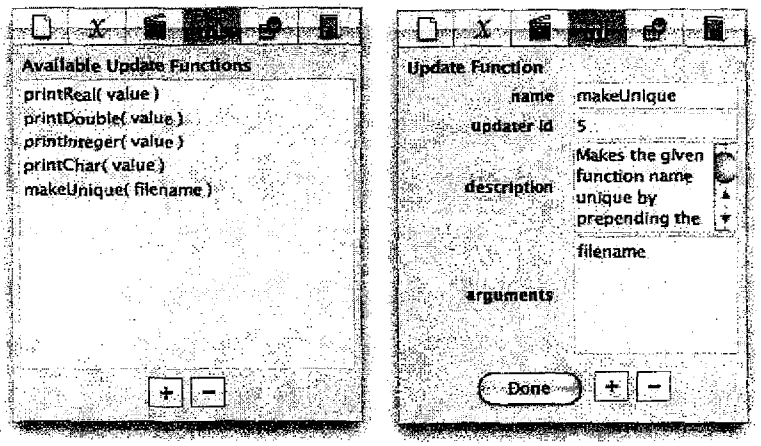

Figure 145. The update function list (left) and function details (right).

Additional update functions can be added by clicking the add button and entering the details of the update function as shown in Figure 145 (right).

BIBLIOGRAPHY

Aho, A. V. and J. D. Ullman. 1972. *The theory of parsing, translation and compiling, Vol. II: Compiling.* Prentice Hall.

Ahuja, L. R., O. David, and J. C. Ascough II. 2004. Developing natural resource models using the object modeling system: Feasibility and challenges. Meeting Proceedings. International Environmental Modelling and Software Society (Iemss) 2004 Conference - Complexity and Integrated Resources Management. Osnabruck, Germany.

Akarsu, E., F. Fox, W. Furmanski, and T. Haupt. 1998. WebFlow-high-level programming environment and visual authoring toolkit for high performance distributed computing. in Proceedings of Supercomputing '98: High Performance Networking and Computing. IEEE Computer Society. 1-7.

Ambrose, R. B. Jr., T. A. Wool, and J. L. Martin. 1993. The water quality analysis simulation program, WASP5, Part A: Model documentation. U.S. Environmental Protection Agency: Athens, GA.

Armstrong, C., R. W. Ford, J. R. Gurd, M. Lujan, K. R. Mayes, and G. D. Riley. 2005. Performance control of scientific coupled models in grid environments. *Concurrency and Computation: Practice and Experience.* 17(2-4): 259-295.

Armstrong, R., D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. 1999. Toward a common component architecture for high-performance scientific computing. in Proceedings of the Conference on High-Performance Distributed Computing.

Balaji, V. 2002. The FMS manual: A developer's guide to the GFDL Flexible Modeling System. Internet. Available from: http://www.gfdl.noaa.gov/~vb/FMSManual; accessed 1 June 2006.

Beckman, P. H., P. K. Fasel, W. F. Humphrey, and S. M. Mniszewski. 1998. Efficient coupling of parallel applications using PAWS. in Proceedings of the 7th

IEEE International Symposium on High Performance Distributed Computing. Chicago, IL. 215-222.

Benson, D. 2006. JGraph and JGraph Layout Pro User Manual. Internet. Available from http://www.jgraph.com/pub/jgraphmanual.pdf; accessed 1 June 2006.

Benz, J., R. Hoch, and T. Legovic. 2001. ECOBAS - Modelling and documentation. *Ecological Modelling.* 138: 3-15.

Bettencourt, M. T. 2002. Distributed model coupling framework. in Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing. Edinburgh, Scotland. 284- 290.

Beven, K. J. 1997. Topmodel: A critique. *Hydrological Processes.* 11(9): 1069-1085.

Blackmon, M., B. Boville, F. Bryan, R. Dickinson, P. Gent, J. Kiehl, R. Moritz, D. Randall, J. Shukla, S. Solomon, G. Bonan, S. Doney, I. Fung, J. Hack, E. Hunke, J. Hurrell, J. Kutzbach, J. Meehl, B. Otto-Bliesner, R. Saravanan, E. K. Schneider, L. Sloan, M. Spall, K. Taylor, J. Tribbia, and W. Washington. 2001. The Community Climate System Model. *Bulletin of the American Meteorological Society.* 82(11): 2357-2376.

Blind, M. W., A. Ubbels, L. R. Wentholt, Th. L. van Stijn, A. H. Bakema, J. D. Bulens, J. J. Noort, B. van Adrichem, J. Stout, and F. C. van Geer. 2000. Towards a well-oiled model infrastructure for water management: the generic framework water program. in Proceedings of HydroInformatics 2000. Cedar Rapids, IA.

Breunese, A. P. J., J. L. Top, J. F. Broenink, and J. M. Akkermans. 1998. Libraries of reusable models: Theory and application. *Simulation.* 71(1): 7-22.

Bulatewicz, T. and J. Cuny. 2005. Interface-based support for model coupling: Spatial representation and compatibility issues. in Proceedings of the 8th International Conference on GeoComputation. Ann Arbor, MI.

Bulatewicz, T., J. Cuny, and M. Warman. 2004. The potential coupling interface: Metadata for model coupling. in Proceedings of the 2004 Winter Simulation Conference. Washington D.C. 1: 175-182.

Chamberlain, B. L., S. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby. 2000. ZPL: A machine independent programming language for parallel computers. *IEEE Transactions on Software Engineering.* 26(3): 197-211.

Dagum, L. and R. Menon. 1998. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering.* 5(1): 46-55.

Dahmann, J. S., R. M. Fujimoto, and R. M. Weatherly. 1998. The DoD high level architecture: An update. in Proceedings of the 1998 Winter Simulation Conference. Washington, D.C. 797-804.

DeLong, L. L., D. B. Thompson, and J. K. Lee. 1997. Computer program FourPt: A model for simulating one-dimensional, unsteady, open-channel flow. U.S. Geological Survey Water-Resources Investigations Report 97-4016, Bay St. Louis, Mississippi.

Ding, Y., M. Munch, and M. Laux. 1999. Dynamic coupling of grid-based multidisciplinary applications. 7th Euromicro Workshop on Parallel and Distributed Processing (EUROMICRO PDP' 99). 249- 255.

Doherty, J. E. 2004. PEST Model-independent parameter estimation user manual: 5th Edition. Watermark Numerical Computing, Australia.

Eisenhauer, G. and K. Schwan. 1998. An object-based infrastructure for program monitoring and steering. in Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools. Welches, OR. 10-20.

Essaid, H. I. and B. A. Bekins. 1997. BIOMOC, A multispecies solute-transport model with biodegredation. U.S. Geological Survey Water-Resources Investigations Report 97-4022.

Federal Geographic Data Committee. 1998. Content standard for digital geospatial metadata. Internet. Available from http://www.fgdc.gov/standards/standards_publications; accessed 1 June 2006.

Flerchinger, G. N. 2000. The simultaneous heat and water (SHAW) model: User's manual. Technical Report NWRC 2000-10. USDA-ARS: Boise, ID.

Ford, R. W., G. D. Riley, M. K. Bane, C. W. Armstrong, and T. L. Freeman. 2004. GCF: A general coupling framework. *Concurrency and Computation: Practice and Experience.* 18(2): 163-181.

Fox, M. R., D. C. Brogan, and P. F. Reynolds Jr. 2004. Approximating component selection. in Proceedings of the 2004 Winter Simulation Conference. Washington, D.C. 429-434.

Gijsbers, P. 2003. OpenMI - Harmonizing linkages between water related models. in Proceedings of the International Conference on Application of Integrated Modelling. Tilburg, The Netherlands.

Gooseff, M. N., S. M. Wondzell, R. Haggerty, and J. Anderson. 2003. Comparing transient storage modeling and residence time distribution (RTD) analysis in geomorphically varied reaches in the Lookout Creek Basin, Oregon, USA. *Advances in Water Resources.* 26: 925–937.

Gu, W., G. Eisenhauer, K. Schwan, and J. S. Vetter. 1998. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience.* 10(9): 699-736.

Guo, W. and C. D. Langevin. 2002. User's guide to SEAWAT: A computer program for simulation of three-dimensional variable-density ground-water flow. Techniques of Water-Resources Investigations of the United States Geological Survey. Book 6, Chapter A7.

Haggerty, R. and P. Reeves. 2003. STAMMT-L: Formulation and user's manual. Sandia National Laboratories: Albuquerque, NM.

Hill, L. L., S. J. Crosier, T. R. Smith, and M. Goodchild. 2001. A content standard for computational models. *D-Lib Magazine.* 7(6).

Hill, C., C. DeLuca, V. Balaji, M. Suarez, and A. DaSilva. 2004. The architecture of the earth system modeling framework. *Computing in Science and Engineering.* 6(1): 18-28.

Huber, W. C. and R. E. Dickinson. 1988. Storm water management model - Version 4: User's manual. Technical Report EPA-600/3-88-001a, U.S. Environmental Protection Agency: Athens, Georgia.

Jablonowski, D. J., J. D. Bruner, B. Bliss, and R. B. Haber. 1993. VASE: The visualization and application steering environment. in Proceedings of the 1993 ACM/IEEE conference on Supercomputing. 560-569.

Jobson, H. E. 1989. Users manual for an open-channel streamflow model based on the diffusion analogy. U.S. Geological Survey Water Resources Investigations Report 89-4133.

Jobson, H. E. and A. W. Harbaugh. 1999. Modifications to the diffusion analogy surface-water flow model (DAFlow) for coupling to the modular finite difference ground-water flow model (ModFlow). U.S. Geological Survey Open-File Report 99-217.

Johnson, C. R., S. G. Parker, D. Weinstein, and S. Heffernan. 2002. Component-based problem solving environments for large-scale scientific computing. *Concurrency and Computation: Practice and Experience.* 14(13-15): 1337-1349.

Johnston, R. K., P. F. Wang, H. Halkola, K. E. Richter, V. S. Whitney, B. E. Skahill, W. H. Choi, M. Roberts, R. Ambrose, and M. Kawase. 2003. An integrated watershed-receiving water model for Sinclair and Dyes Inlets, Puget Sound, Washington, USA. Estuarine Research Federation 2003 Conference Estuaries on the Edge: Convergence of Ocean, Land and Culture. Seattle, WA.

Joppich, W., M. Kurschner, and the MpCCI team. 2005. MpCCI - a tool for the simulation of coupled applications. *Concurrency and Computation: Practice and Experience.* 18(2): 183-192.

Knox, R. G., V. L. Kalb, E. R. Levine, and D. J. Kendig. 1997. A problem-solving workbench for interactive simulation of ecosystems. *IEEE Computational Science & Engineering.* 4(3): 52-60.

Kohl, J. A. and P. M. Papadopoulos. 1998. Efficient and flexible fault tolerance and migration of scientific simulations using CUMULVS. Symposium on Parallel and Distributed Tools (SPDT '98). Welches, Oregon.

Krueger, C. W. 1992. Software reuse. *ACM Computing Surveys (CSUR).* 24(2): 131-183.

Larson, J., R. Jacob, and E. Ong. 2005. The model coupling toolkit: A new Fortran90 toolkit for building multiphysics parallel coupled models. *International Journal for High Performance Computing Applications.* 19(3): 277-292.

Leavesley, G. H., P. J. Restrepo, S. L. Markstrom, M. Dixon, and L. G. Stannard. 1996. The modular modeling system - MMS: User's manual. U.S. Geological Survey Open-File Report 96-151.

Leonard, R. A., W. G. Knisel, and D. A. Still. 1987. GLEAMS: Groundwater loading effects of agricultural management systems. *Transactions of the American Society of Agricultural Engineers.* St. Joseph, Michigan. 30(5): 1403-1418.

Lindgren, G. A., G. Destouni, and A. V. Miller. 2004. Solute transport through the integrated groundwater-stream system of a catchment. *Water Resources Research.* 40(3).

Lindlan, K., J. Cuny, A. Malony, S. Shende, B. Mohr, R. Rivenburgh, and C. Rasmussen. 2000. Tool framework for static and dynamic analysis of object-oriented software with templates. in Proceedings of the 2000 ACM/IEEE conference on Supercomputing. 49-59.

McDonald, M. G. and A. W. Harbaugh. 1988. A modular three-dimensional finite difference ground-water flow model. Techniques of Water-Resources Investigations of the United States Geological Survey. Book 6, Chapter A1.

Miller, D. W., J. Guo, E. Kraemer, and Y. Xiong. 2001. On-the-fly calculation and verification of consistent steering transactions. in Proceedings of the 2001 ACM/IEEE conference on Supercomputing. 1-17.

Muralidhar, R. and M. Parashar. 2003. A distributed object infrastructure for interaction and steering. *Concurrency and Computation: Practice and Experience.* 15(10): 957-977.

Neitsch, S. L., J. G. Arnold, J. R. Kiniry, and J. R. Williams. 2001. Soil and water assessment tool (SWAT) user's manual version 2000. Grassland, Soil, and Water Research Laboratory & Blackland Research Center, USDA-ARS: Temple, TX.

Neteler, M. and H. Mitasova. 2004. *Open Source GIS: A GRASS GIS Approach.* Second Edition. Kluwer Academic Publishers, Boston, Dordrecht.

Ninnemann, Jeff. 2004. Summary report: Basin wide hyporheic parameter estimates for the H.J. Andrews Experimental Forest, Oregon. Technical Report. Oregon State University.

Page, E. H., A. Buss, P. Fishwick, K. J. Healy, R. E. Nance, and R. J. Paul. 1998. The modeling methodological impacts of web-based simulation. in Proceedings of the 1998 SCS International Conference on Web-Based Modeling and Simulation. San Diego, CA, 11-14 January. 123-128.

Parnas, D. L. 1972. On the criteria for decomposing systems into modules. *Communications of the ACM.* 15(12): 1053-1058.

Piacentini, A. and the PALM group. 2002. PALM: A dynamic parallel coupler. in Proceedings of High Performance Computing for Computational Science - VECPAR 2002: 5th International Conference. 479-492.

Plentinger, M. C. and F. W. T. Penning de Vries, eds. 1996. CAMASE, Register of agro-ecosystems models. Internet. Available at http://library.wur.nl/camase/; accessed 1 June 2006.

Rajlich, V. and N. Wilde. 2002. The role of concepts in program comprehension. in Proceedings of the 2002 International Workshop on Program Comprehension. IEEE Computer Society Press, Los Alamitos, CA. 271-278.

Rasmussen, C. E., K. A. Lindlan, B. Mohr, and J. Striegnitz. 2001. CHASM: Static analysis and automatic code generation for improved Fortran 90 and C++ interoperability. Los Alamos Computer Science Institute 2001 Symposium. Santa Fe, NM.

Rathmayer, S. and M. Lenke. 1997. A tool for on-line visualization and interactive steering of parallel HPC applications. in Proceedings of the 11th International Parallel Processing Symposium, IPPS 97. 181-186.

Robinson, S., R. E. Nance, R. J. Paul, M. Pidd, and S. J. E. Taylor. 2004. Simulation model reuse: Definitions, benefits and obstacles. *Simulation Modelling Practice and Theory.* 12: 479-494.

Ross, M., J. Geurink, A. Aly, P. Tara, K. Trout, and T. Jobes. 2004. Integrated hydrologic model (IHM) Volume 1: Theory manual. Tampa Bay Water and Southwest Florida Water Management District.

Rowan, A. 2001. Development of the multiple model broker, a system integrating stormwater and groundwater models of different spatial and temporal scales using embedded GIS functionality. Ph.D. diss. Rutgers, The State University of New Jersey: New Brunswick, NJ.

Runkel, R. L. 1998. One-dimensional transport with inflow and storage (OTIS) - A solute transport model for streams and rivers. U.S. Geological Survey Water-Resources Investigations Report 98-4018.

Schaffranek, R. W., R. A. Baltzer, and D. E. Goldberg. 1981. A model for simulation of flow in singular and interconnected channels. Techniques of Water-Resources Investigations of the United States Geological Survey. Book 7, Chapter C3.

Shengsheng, Y., W. Yuanqiao, H. Liwen, and D. Jian. 2005. Framework of distributed numerical model coupling system. 9th IEEE International Symposium on Distributed Simulation and Real-Time Applications. 187-194.

Simon, E. Le Select Tutorial. INRIA-Rocquencourt. Internet. Available from http://www-caravel.inria.fr/~leselect; accessed 1 June 2006.

Sklower, K., H. Robinson, C. R. Mechoso, L. A. Drummond, J. Spahr, J. D. Farrara, and E. Mesrobian. The distributed data broker: A decentralized mechanism for periodic exchange of fields between multiple ensembles of parallel computations. Internet. Available from http://www.atmos.ucla.edu/~mechoso/esm/ddb_pp.html; accessed 1 June 2006.

Smith, P., D. S. Powlson, J. U. Smith, and P. Falloon. 1997. SOMNET: A global network and database of soil organic matter models and long-term experimental datasets. *The Globe*. 38: 4-5.

Sottile, M. 2001. The design of a general method for constructing coupled scientific simulations. Masters thesis. University of Oregon.

Storey, M., F. Fracchia, and H. Müller. 1997. Cognitive design elements to support the construction of a mental model during software visualization. in Proceedings of the 5th International Workshop on Program Comprehension, Dearborn, MI. 17-28.

Storey, M., K. Wong, F. Fracchia, and H. Müller. 1997. On integrating visualization techniques for effective software exploration. in Proceedings of IEEE Symposium on Information Visualization (InfoVis'97). Phoenix, AZ. 38-45.

Swain, E. D. and E. J. Wexler. 1996. A coupled surface-water and ground-water flow model (ModBranch) for simulation of stream-aquifer interaction. Techniques of Water-Resources Investigations of the United States Geological Survey, Book 6, Chapter A6.

Sydelko, P. J., K. A. Majerus, J. E. Dolph, and T. N. Taxon. 1999. A dynamic object-oriented architecture approach to ecosystem modeling and simulation. In Proceedings of the 1999 American Society of Photogrammerty and Remote Sensing (ASPRS) Annual Conference. 410-421.

Tarboton, D. G. and C. H. Luce. 1996. Utah energy balance snow accumulation and melt model (UEB), computer model technical description and users guide. Utah Water Research Laboratory and USDA Forest Service Intermountain Research Station.

Trescott, P. C., G. F. Pinder, and S. P. Larson. 1980. Finite-difference model for aquifer simulation in two dimensions with results of numerical experiments. In Techniques of Water-Resources Investigations of the United States Geological Survey, Book 7, Chapter C1.

Valcke, S., A. Caubel, R. Vogelsang, and D. Declat. 2004. OASIS3 Ocean Atmosphere Sea Ice Soil User's Guide Technical Report TR/CMGC/04/68, CERFACS, Toulouse, France.

Valcke, S., E. Guilyardi, and C. Larsson. 2005. PRISM and ENES: a European approach to Earth system modelling. *Concurrency and Computation: Practice and Experience*. 18(2): 247-262.

van Wijk, J. and R. van Liere. 1997. An environment for computational steering. in G.M. Nielson, H. Muller, and H. Hagen, eds, *Scientific Visualization: Overviews, Methodologies, and Techniques*. Computer Society Press. 89-110.

Vetter, J. and K. Schwan. 1997. High performance computational steering of physical simulations. in Proceedings of the 11th International Parallel Processing Symposium, IPPS 97. 128-132.

Vetter, J. and K. Schwan. 1995. Progress: A toolkit for interactive program steering. in Proceedings of the 24th International Conference on Parallel Processing. 2:139-142.

Whelan, G., K. J. Castleton, J. W. Buck, B. L. Hoopes, M. A. Pelton, D. L. Strenge, G. M. Gelston, and R. N. Kickert. 1997. Concepts of a framework for risk analysis in multimedia environmental systems (FRAMES). PNNL-11748, Pacific Northwest National Laboratory, Richland, Washington.

WRIA1. The Watershed Management Plan. Internet. Available from http://www.wria1project.wsu.edu/watershedplan.htm; accessed 1 June 2006.