UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Paul J. Iglinski

TITLE OF THESIS: An Execution Replay Facility and Event-Based Debugger for the *Enterprise* Parallel Programming System

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1994

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly, or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Paul J. Iglinski
10278 - 87 Street
Edmonton, AB
T5H 1N6

Date_____

UNIVERSITY OF ALBERTA

An Execution Replay Facility and Event-Based Debugger for the *Enterprise* Parallel Programming System

by

Paul J. Iglinski

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science

Department of Computing Science

Edmonton, Alberta
Fall, 1994

Abstract

Enterprise is an object-oriented Parallel Programming System for distributed computing. Under the control of a graphical interface, the system provides a complete programming environment for parallel distributed computing on a network of heterogeneous workstations. This environment supports designing, coding, compiling, executing, performance monitoring, debugging, and maintaining distributed programs. The *Enterprise* model provides templates for different types of parallelization techniques to be applied to modules of standard sequential code. All the appropriate communication and synchronization code is automatically inserted into the user's code during compilation.

The research conducted for this thesis was predicated upon the hypothesis that execution replay and event-debugging can be seamlessly integrated into both the *Enterprise* model and Parallel Programming System. The thesis presents such an execution replay facility and an event-based breakpointing facility for debugging *Enterprise* programs during replay. These facilities allow the user to perform high-level debugging by setting conditional breakpoints on message-passing events. Sequential debuggers can then be attached where needed on individual processes for low-level debugging. The breakpointing and replay are coupled with a graphical animation of the program to provide the user with an intuitive macroscopic view of the program's execution. Interactive controls allow the user to stop at any time, examine recorded parameters, modify breakpoints, resume execution, or abort execution. The thesis establishes that the *Enterprise* model is consistent with the concept of execution replay and event-level debugging.

Acknowledgments

The work presented in this thesis is part of a large team effort by the members of the *Enterprise* research group in the Department of Computing Science at the University of Alberta. At every stage of this research and development, the ideas and suggestions of many individuals in this group were incorporated into this debugging facility.

In particular, I would like to thank my co-supervisors, Dr. Duane Szafron and Dr. Jonathan Schaeffer, for their guidance and patience throughout this endeavor. The direction and design of this thesis research owes much to their direct involvement. My appreciation is extended, as well, to the other examiners of my thesis, Dr. Werner Jeorg and Dr. John Buchanan, for their suggestions and corrections which have been incorporated into the final version. Thanks also go to Dr. John Tartar, who chaired my thesis defense.

I would also like to acknowledge the special contributions of various members of the *Enterprise* team: Greg Lobe, for his early help with facilitating my understanding of the original Smalltalk interface for *Enterprise*; Randal Kornelsen, for his explanations of the intricacies of the code for the *Enterprise* executive; Ian Parsons, for his on-going involvement with integrating my work into the ever evolving *Enterprise* system; Christopher Morrow, for his version control work and cooperation in porting the interface to VisualWorks 1.0; Diego Novillo, for his modifications to the *Enterprise* compiler; Dr. Zhonghua Yang, for his valuable help in directing me to many of the leading articles in the field of distributed debugging; and all the other recent members of the *Enterprise* team, Steve MacDonald, David Woloschuk, Carol Smith, Kent McPhee, and Stan Melax, for their suggestions and encouragement. My thanks go to all involved.

This research would not have been possible without the financial support of the Province of Alberta, by way of a Province of Alberta Graduate Scholarship; the Department of Computing Science at the University of Alberta, by way of Graduate Teaching Assistantships; and my supervisors, Drs. Szafron and Schaeffer, by way of grants from research funds at their disposal.

Finally, my appreciation and thanks go to my wife Monika and my sons Steven and Peter for their support, understanding, and forbearance during these past two years of graduate studies.

Table of Contents

Chapter 1 Introduction 1
- 1.1. *Enterprise* Motivations 1
- 1.2. The Goals of *Enterprise* 1
- 1.3. Thesis Motivation 3
- 1.4. Thesis Implementation Goals 3
- 1.5. Scope and Organization of the Thesis 4

Chapter 2 The *Enterprise* Parallel Programming System 6
- 2.1. Overview 6
- 2.2. The Meta-Programming Model 7
  - 2.2.1. The Enterprise Asset 7
  - 2.2.2. The Individual Asset 7
  - 2.2.3. The Receptionist Asset 8
  - 2.2.4. The Representative Asset 8
  - 2.2.5. The Department Asset 8
  - 2.2.6. The Line Asset 8
  - 2.2.7. The Division Asset 9
  - 2.2.8. The Service Asset 9
- 2.3. The Programming Model 11
  - 2.3.1. p-Calls, f-Calls, Futures, and Lazy Synchronization 11
  - 2.3.2. Ordered and Unordered Replicated Assets 12
  - 2.3.3. Parameter Passing Macros 13
  - 2.3.4. Programming Restrictions 14
- 2.4. The *Enterprise* User Interface 14
  - 2.4.1. The Design View 15
  - 2.4.2. Animation View 22
- 2.5. The *Enterprise* Pre-Compiler 25
- 2.6. The Communication Manager 26
- 2.7. The *Enterprise* Executive 26
- 2.8. Related Work 29
  - 2.8.1. HeNCE 29
  - 2.8.2. Parallel Computational Frames—PCF 30

Chapter 3 The Art and Science of Distributed Debugging 31
3.1. The Difficulties of Distributed Debugging 31
3.2. A Model for Distributed Debugging 33
3.3. Techniques for Distributed Debugging 36
3.4. Incorporating Debugger Methodologies and Approaches 38
3.5. Related Debuggers 40
3.5.1. Instant Replay 40
3.5.2. DPD—A Distributed Program Debugger 41

Chapter 4 The *Enterprise* Replay Facility 43
4.1. Overview 43
4.2. Ensuring the Outcome of Race Conditions 43
4.3. The Replay Algorithm 51
4.4. Modifications to the *Enterprise* Executive 52
4.4.1. Communications during Replay 52
4.4.2. Message-Passing Events 52
4.4.3. Die Events 54
4.5. Extensions to the *Enterprise* User Interface 54
4.5.1. The Design View 54
4.5.2. The Replay View 58
4.5.2.1. Main Menu 58
4.5.2.2. Asset Menu: Sequential Debuggers 59
4.5.2.3. Message Queue Menu: Message Queue Lists 60

Chapter 5 Breakpointing in Replay Mode 61
5.1. Overview 61
5.2. Event Breakpoints 61
5.3. The Breakpoint Browser 62
5.3.1 Navigation Tools 63
5.3.1.1. Assets List 63
5.3.1.2. Event Type List 64
5.3.1.3. Collaborators List 65
5.3.1.4. Breakpoints List 66
5.3.2. Definition Tools 67
5.3.2.1. The Encounter Setting 67
5.3.2.2. Constructing Conditions 67
5.3.2.2. Adding and Removing Conditions 69
5.3.3. Accepting Breakpoints 69
5.3.4. Editing Breakpoints 69
5.4. Graphical Display of Breakpoints 70

5.4.1. Description of Symbols 70
5.4.2. Highlighting during Breakpoint Setup 71
5.4.3. Graphical Control of the Breakpoint Browser 73
5.5. Breakpoint Triggering 74
5.6. Implementation Issues 75

Chapter 6 Conclusion 76
6.1. Individual vs. Group Contributions 76
6.2. Future Work 77
6.3. Scalability 78
6.4. Conclusion 79

Bibliography 81

Appendix A New Enterprise File Formats 85
A.1. Event Log File 85
A.1.1. Notation 85
A.1.2. Syntax 85
A.1.3. Semantics 87
A.1.4. Examples 90
A.2. Default Format and Format Files 92
A.2.1. Notation 92
A.2.2. Syntax 92
A.2.3. Semantics 92
A.2.4. Examples 93

Appendix B Unimplemented Features and Identified Problems 95
B.1. Unimplemented Features 95
B.1.1. Parameter Logging Editor 95
B.1.2. Breakpointing 95
B.1.3. Sequential Debugging 96
B.2. Identified Problems 96
B.2.1. Animation during Replay 96
B.2.2. Communication Managers 96
B.2.3. Persistence of Breakpoints 97

List of Figures

2.1. A fully expanded view of a complex *Enterprise* program 10
2.2. A complex *Enterprise* program progressively expanded 16
2.3. The meta-program for **`CubeSquare`** 17
2.4. The code for **`CubeSquare`** 18
2.5. Replication Dialog Box 19
2.6. Compile Dialog Box 20
2.7. Run Dialog Box 21
2.8. Asset Options Dialog Box 22
2.9. **`CubeSquare`** animating 23
2.10. Message Queue List 24
2.11. State transition diagram for asset states 25
2.12. **`CubeSquare`** process configuration 28

3.1. Model of a Distributed Debugging Environment 34
3.2. Decomposition of a Distributed Debugging System (DDS) 35

4.1. A simple Line and simple Department 44
4.2. Non-determinism #1—Meta-program for **`ManShootsDog`** 45
4.3. Program code for **`ManShootsDog`** 46
4.4. Non-determinism #2—Meta-program for **`BadMath`** 47
4.5. Program code for **`BadMath`**, replicas having static variables 48
4.6. Non-determinism #2—Meta-program for **`UncertainDivision`** 49
4.7. Program code for **`UncertainDivision`** 50
4.8. Modified `Divide` function with rounding to create non-determinism 51
4.9. Parameter Logging Editor 56

5.1. Breakpoint Browser 63
5.2. Replay View of **`CubeSquare`** with assets selected 71
5.3. Replay View of **`CubeSquare`** with assets and event type selected 72
5.4. Replay View of **`CubeSquare`** with assets, event type, and collaborator selected 72
5.5. Replay View with breakpoints set 73

Chapter 1

Introduction

This thesis constitutes a part of *Enterprise*, an object-oriented Parallel Programming System (PPS) for distributed computing under development in the Department of Computing Science at the University of Alberta. *Enterprise* is an on-going research effort spanning several years and involving the active participation of more than a dozen researchers and developers. The specific contribution of this thesis is the development and implementation of an execution replay facility and an event-based breakpointing facility for debugging during replay.

1.1. *Enterprise* Motivations

With today's proliferation of powerful workstations linked by increasingly fast, reliable, and economical networks with ever increasing bandwidths, a phenomenal amount of "free" computing power has become available to programmers in organizations, both large and small. Together these resources would constitute a super-computing facility if only the tools and methods were available for easily tapping into this potential of idle CPU cycles. Many computationally intensive applications that require days on a single processor could potentially execute in a matter of hours on a room full of workstations that would normally be under-utilized. What is needed is a set of easy-to-use tools which bridge the gap between the sequential and parallel worlds. Such a tool set should provide some well managed parallelization techniques while shielding the user from low-level details of hardware requirements and communication protocols. Mechanisms for transforming sequential code into parallel code should be largely automatic, requiring the user to make only minimal code changes himself.

1.2. The Goals of *Enterprise*

The *Enterprise* project is designed with the following objectives:

- To provide a complete programming environment for distributed computing on a network of workstations which supports designing, coding, compiling, executing, performance monitoring, debugging, and maintaining distributed programs.

- To provide an easy-to-use, familiar, high-level programming model that insulates the user from the details and complexities of inter-process communication and hardware concerns among heterogeneous machines on a network.

- To provide a high-level, intuitive, and readily comprehensible graphical interface to a complex and potentially confusing parallel architecture. All the programming tools have a uniform style in the interface, resulting in a reduced learning curve for users.

- To ensure reliability and minimize the introduction of errors due to hazards of parallelization—deadlock, synchronization, and non-determinacy.

- To support commonly used sequential languages, presently C and eventually Fortran, without syntactic modifications or calls to external library routines.

- To facilitate the coarse-grained parallelization of existing software with a minimal amount of re-coding.

- To reduce the cost of parallel software development on existing resources.

Enterprise takes a bold step towards bridging the chasm between sequential and parallel computing. It makes compromises, in the course of achieving these goals, between usability and efficiency, between reliability and versatility, between high-level abstraction of design and low-level user control. The transition is not unlike the paradigm shift between coding in assembler and coding in a high-level symbolic language. Just as a skilled assembler programmer might produce more efficient code than a C programmer, even with compiler optimizations, an experienced parallel programmer manipulating sockets on a network can produce a more efficient program than an *Enterprise* user. But when one examines the software engineering issues of programmer training time, program development time, program debugging time, and program maintainability, the true costs of a lower level solution become apparent.

Indeed, the goals of *Enterprise* are not easily attainable. Yet without their realization, the benefits of parallel computational power will remain the esoteric prerogative of a chosen few who have become adept at facing the challenges of low-level communication packages and the pitfalls of unmanaged parallelism.

Enterprise is not intended to be a low-budget substitute for multi-million dollar Crays, vector processors, or massively parallel connection machines. Fine- or medium-grained parallelism will not work efficiently under *Enterprise*. The granularity must be sufficiently large to balance out the relatively high cost of message passing over a network. Users whose paramount concern is minimizing CPU cycles will likely not be satisfied. *Enterprise* does incur overheads for process management. Moreover, certain

kinds of problems might not have parallel solutions easily mapped to the *Enterprise* model. Shared memory, for instance, can be simulated in *Enterprise*, but the frequent access of such shared variables may prove overly costly. Likewise, peer-to-peer communication is not presently supported.

Nevertheless, there is a potentially large user community for a parallelization system such as *Enterprise*. One such community is scientific researchers with large computationally intensive problems having independent large-grained components. Another community is computing science education. *Enterprise* has already been used as a vehicle for teaching in an introductory course in parallel computing [SS94b].

1.3. Thesis Motivation

While the *Enterprise* model eliminates the possibility of inter-process deadlock and reduces the possibility of synchronization errors and non-deterministic behaviors, debugging an *Enterprise* program, just like any other parallel program, can still be a formidable task without some specialized, high-level debugging tools.

The *Enterprise* system allows the user to log "major events" during any execution of an *Enterprise* program. This log contains sufficient information on all message passing events, as well as important state changes, to describe a deterministic execution path. With a post-mortem animation view, the user can simulate this particular execution and stop at any point to examine parameter and return values captured in message-passing events logged in the event file. But if the file is of any significant size, it soon becomes impractical to step through it examining the captured parameter values. Any unlogged parameters or internal process variables are completely inaccessible.

Herein lies the motivation for a genuine execution replay mechanism coupled with a high-level event-based breakpointing facility which allows the user to deterministically re-execute the program and stop when some specified condition is encountered. When a set breakpoint is encountered, the user can directly examine any logged parameter in the inter-process messages or attach a sequential debugger to selected processes to probe any unlogged parameters or the processes' internal states. To further facilitate the debugging task, a special menu based editor is also provided for specifying the parameters to be logged during a program run.

1.4. Thesis Implementation Goals

This thesis is predicated the hypothesis that execution replay and event-based debugging can be seamlessly integrated into the Enterprise model and programming system.

The implementation goals of this thesis are twofold:

1. To develop an execution replay facility for *Enterprise* under the control of the interface in order to resolve the problem of non-determinism during debugging. This facility should be fully integrated with the existing interface, in particular with the program animation tools.

2. To develop a high-level event-based debugger built upon the replay facility. The debugger should have the following characteristics:

   - An effective means of setting multiple event-level breakpoints which utilizes information captured in the event log.

   - User control over which parameter and return values are captured when an event log is generated.

   - Full integration with the pre-existing *Enterprise* programming environment. This includes adhering to the model, style, control conventions, and the look and feel of the *Enterprise* interface.

   - A facility that is easy to learn and use. In a unified environment the effort of learning new tools is amortized.

   - Complete coordination of graphical displays with the breakpointing mechanisms. This includes graphical control over breakpoint definition and selection.

   - Easy accessibility to sequential debuggers. Attaching sequential debuggers during execution replay to selected processes should involve just a simple menu selection.

The successful attainment of these goals establishes that the *Enterprise* model is consistent with the concept of execution replay and event-level debugging.

1.5. Scope and Organization of the Thesis

At the core of this thesis is the actual implementation of an execution replay mechanism and breakpointing facility for *Enterprise*. The design and implementation of a practical working system within the existing *Enterprise* model were the primary focus of this research. A survey of distributed debugging technology provided the theoretical foundation for this work. The exploration of theoretical questions surrounding this area of research, however, was not within the mandate of this thesis. Likewise, no attempt has been made to develop metrics to evaluate the usability of the facility.

This thesis is organized as follows:

- Chapter 2 provides an overview of *Enterprise*, excluding the facilities developed in this thesis.
- Chapter 3 examines the general issues involved in distributed debugging and looks at other approaches to the problem in similar systems.
- Chapter 4 describes the execution replay facility and how it was implemented.
- Chapter 5 describes the breakpointing facility and how it was implemented.
- Chapter 6 summarizes the thesis, describing unimplemented features, identified problems, and future developments envisioned for debugging in *Enterprise*.

Chapter 2

The *Enterprise* Parallel Programming System

2.1. Overview

Enterprise [SSL93] [SSI94] is a fully integrated programming environment which allows the user to design, code, compile, execute, monitor, profile, analyze, test, and debug parallel programs in a distributed environment of heterogeneous UNIX workstations. A uniform graphical user interface provides the mechanisms for modularizing the program components and selecting the desired communication techniques. All the necessary communication and synchronization code for parallel execution on a selected group of workstations is automatically inserted into the user's code when the program is compiled. A C programmer need not learn any new language, language constructs, or libraries of special functions for handling inter-process communication or heterogeneity concerns. A simple menu-based graphical meta-programming model for expressing parallelism is really the only new symbolic construct for the programmer to learn. Learning this model, together with how to use the tools and features provided by the "intuitive" and "friendly" *Enterprise* interface, maps onto a fast and flat learning curve.

The *Enterprise* model provides the user with templates to implement various parallelization techniques. As such, it can be categorized as a template-based PPS. Arbitrary communication patterns among concurrent processes are not allowed. The process communication graph is determined by the selected templates.

The *Enterprise* system consists of four distinct components: a pre-compiler, the *Enterprise* executive, a communication manager, and the object-oriented user interface. The user provides a fifth component, an *Enterprise* program: a modularized C program together with any user-required libraries, a meta-program for describing the parallelism desired, and lists of machines to use during execution.

Before describing the roles of these components, it is instructive to first describe the meta-programming model at the heart of *Enterprise*.

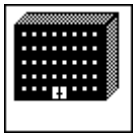
2.2. The Meta-Programming Model

The *Enterprise* model employs an analogy with a business organization to represent, using intuitive terminology, different types of parallelization techniques. Business enterprises inherently manifest parallelisms, both synchronous and asynchronous. There is an intrinsic similarity between the units of a business organization (departments, divisions, and factory assembly lines all ultimately composed of individual workers) and the groups of individual processes executing in a distributed computing environment. This analogy provides us with a consistent terminology and contributes to a simplified conceptualization of a complex architecture.

Every program entity in *Enterprise*, be it an individual module or integrated group of modules, is called an *asset*. An asset can be of various types: an Enterprise, an Individual, a Receptionist, a Representative, a Line, a Department, a Division, or a Service. Lines, Departments, and Divisions are composite assets representing different parallelization techniques. Many of these asset types can be parallelized further by replication so that multiple copies of the asset can be working on a task concurrently. Some asset types can be converted to other types with different parallelization properties. Some can be extended through the addition of assets.

The following descriptions of *Enterprise* assets are accompanied by their icons as they appear on the screen.

2.2.1. The Enterprise Asset



The Enterprise asset is a composite that represents the entire user program. It is analogous to an entire business organization. An Enterprise contains a single asset: an Individual, a Line, or a Department. Neither the Enterprise asset itself nor the single asset within it can be replicated. Code is not directly associated with an Enterprise asset; code belongs to its codable components.

2.2.2. The Individual Asset

An Individual is analogous to an individual worker in a business enterprise. Individuals are codable assets. Each individual corresponds to a module of C code which executes sequentially as a single process, unless of course, it happens to be assigned to an actual parallel machine. When an Individual is called, it works to completion before it can accept new work in the form of another asset call. In general, Individuals can be replicated. They can also be converted to Lines, Departments, or Divisions.

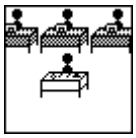
2.2.3. The Receptionist Asset

A Receptionist can be viewed as a specialized kind of Individual. It is a codable asset. The analog of a Receptionist is a receptionist in a business unit whose primary responsibility is to direct incoming requests to the appropriate sub-unit. It appears as the first asset in every Line, Department, and Division. As such, it constitutes the entry point to these composite assets. It shares its name with the composite asset to which it belongs. Since each asset can have only a single entry point, a Receptionist cannot be replicated explicitly. If, however, the composite itself is replicated, each replica will have its own single Receptionist. Receptionists cannot be converted to any other asset type.

2.2.4. The Representative Asset

A Representative, another kind of specialized Individual, is used only in Divisions as a leaf node in a parallel recursion tree. It, too, is a codable asset. It can be replicated or converted to a Division having the same name. The name of a Representative is the same as the name of the Division to which it belongs.

2.2.5. The Department Asset



A Department's counterpart in the business world is a department with a receptionist directing incoming requests to the appropriate sub-unit. A Department is a composite asset that contains, aside from the receptionist, one or more heterogeneous component assets. These component assets can be Individuals, Lines, Divisions, or other Departments. The components can be called only by the Receptionist. If the Department itself is nested in a Line, the Receptionist or any of the components can call out of the Department to the next asset in the Line. Components may be added, deleted, replicated, or converted to other qualified asset types. A Department itself can be replicated. No code is directly associated with a Department.

2.2.6. The Line Asset

A Line is like an assembly line in a factory, or a pipeline in computing science terminology. Like a Department, it is a composite asset with a Receptionist at its entry point and one or more component assets: Individuals, Departments, Divisions, or other Lines. Each asset in a Line, including the Receptionist, contains a call to the next asset in the Line. When an asset in a Line finishes its task, it is ready to receive another call to do more work. This behavior resembles an assembly line where one worker performs a specialized task on a product, passes it on to the next worker, and then

immediately takes a new item to process. Bottlenecks can occur in a Line if one asset receives work faster than it can process it. Replication can be used on any component, barring the Receptionist, of course, to alleviate this congestion. Components can be added, deleted, replicated, and converted to other legal asset types. Lines themselves can be replicated. If a Line is embedded in another Line, only the last asset of the inner Line can call out to the next asset of the outer Line. Lines are not codable.

2.2.7. The Division Asset

A Division is the only recursive *Enterprise* asset. It can be used to implement parallel recursion for divide-and-conquer algorithms. A Division is a composite asset which consists of a Receptionist and a single component, which is either a Representative or another Division. All the assets in a Division share the same name. Moreover, Receptionists and Representatives share the same code. The code for a Receptionist is transformed by *Enterprise* into a parallel recursive call, while the Representative executes sequential recursion. Divisions and Representatives can be replicated and converted into each other, except at the top level of the Division. Thus the user can control the depth of the parallel recursion tree by creating nested levels of Divisions. The branching factor, or fan-out, at each level is the replication factor selected for the Representative or Division at that level. If a Division is nested in a line, any asset in the Division can call outside to the next asset in the Line. Divisions are not codable assets.

2.2.8. The Service Asset

A Service is an asset that can be called by any other non-Service asset in the program. It is comparable to an individual resource, such as a photocopier in a business. It can be used by everyone on a first-come-first-served basis. Services do not contain any other assets and they cannot be replicated. Services can be used, for example, to simulate shared memory (global variables) or to control access to files shared by multiple assets concurrently. Like Individuals, they are codable assets.

The mechanisms for converting assets from Individuals to composite types and for adding assets to Lines and Departments provide the means for creating programs of arbitrary complexity—multiple levels of assets nested within assets. Figure 2.1 shows a fully expanded view, displaying all the codable assets, of the meta-program of a complex *Enterprise* program. It consists of a Program Pane with a Line (`LineA`), whose Receptionist (also called `LineA`), is the program's first codable asset. The next component is a Department (`DepartmentA`), made up of a Receptionist (also called `DepartmentA`), an Individual (`IndividualA`), a Line (`LineB`), a Division (`DivisionA`), and another Department (`DepartmentB`). The last asset in `LineA` is yet another Individual (`IndividualC`). Two Services (`ServiceA` and `ServiceB`) are visible in the Service

Pane at the top. Though such a complex asset diagram is unlikely in practice, this example illustrates the flexibility of program design in *Enterprise*. Despite its complexity, an asset diagram such as this is easy to build and easy to modify. Typing in the asset names is actually the most time-consuming task.

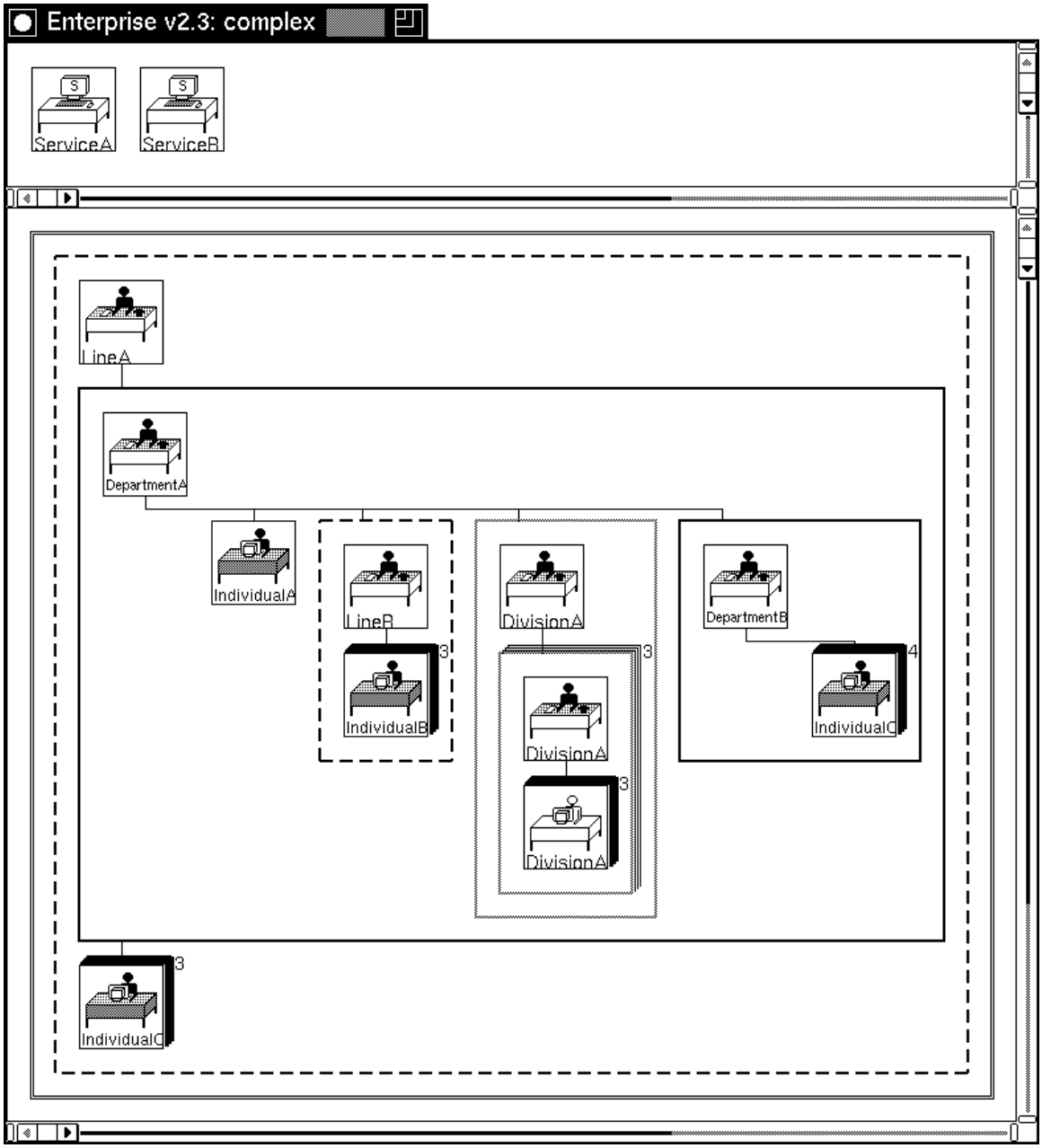


Figure 2.1. A fully expanded view of a complex *Enterprise* program.

2.3. The Programming Model

The programming language for *Enterprise* is standard C with a few qualifications. Only the codable asset types, Individuals, Receptionists, Representatives, and Services, have code directly associated with them. Each codable asset in the program must have a unique name, except for the Receptionists and Representative nested in a Division, which all share the same name and code. Each of these assets is a module unto itself, resembling an Abstract Data Type (ADT), in that each asset has a set protocol for being accessed (a single entry point) and its internal procedures are encapsulated, inaccessible from other modules. At the meta-programming level, composite assets, Departments, Lines, and Divisions, also embody the ADT principle of encapsulation with a Receptionist as the unique interface to the outside. If any functions are to be shared between modules, they must be defined in separate files in a special directory[1] which *Enterprise* maintains, compiles, and links with a compiled program. As with "pure" ADTs, global variables intended to be visible across assets are not allowed. However, globally visible information can be managed through Services. Note that a global variable whose "scope" is the group of functions in a single asset can be declared "globally" within that asset's module. Such asset-level globals are never accessed by *Enterprise* code or other asset code.

2.3.1. p-Calls, f-Calls, Futures, and Lazy Synchronization

Enterprise differentiates between two types of asset calls, *p-calls* and *f-calls*. A *p-call* is like a procedure call with no side effects on the parameters and no return value. If `AssetA` calls `AssetB` using a p-call, `AssetA` continues to execute concurrently with `AssetB` without any need ever to synchronize their behaviors. This is purely asynchronous parallelism. An *f-call*, however, is like a function call where a return value is expected either directly or via side effects to the parameters. When an f-call is made, the caller does not block immediately. Rather, the caller continues to execute until the results of the asset call are actually needed. These results are maintained by *Enterprise* in structures called *futures*. If a future is not available when it is needed, the calling asset blocks until the future arrives. This deferred style of synchronization is called *lazy synchronization*. This is, indeed, the central mechanism for achieving parallelism without recourse to system library calls inserted by the user whenever f-calls are made.

Consider the following code segment from `AssetA` with a series of calls to `AssetB`:

---

[1]When a program is created, *Enterprise* sets up a somewhat complex directory structure to manage the multitude of files generated by the user and the *Enterprise* system. The details of this file system are irrelevant to this introduction.

```
int x[5], i, a;

for (i=0; i<5; i++)
      x[i] = AssetB(i);
/*   More code   */
a = x[0];
```

When `AssetB` is first called, the parameter `i` is packed into a message by `AssetA` and sent over the network to the process executing `AssetB`. `AssetA` then immediately continues to execute its code in the for-loop and four more calls are made to AssetA without waiting for return values to be assigned to the array `x`. After the for-loop is completed, `AssetA` continues executing until some value of `x` is directly referenced, as in the assignment to `a`. If the future responsible for this return value has not yet received it from `AssetB`, `AssetA` blocks and waits until the message containing the value arrives. This synchronization preserves the semantics of sequential C.

All the *Enterprise* code for making asset calls and synchronizing on futures is added to the user's code at compile time. At runtime, *Enterprise* maintains all the information on what return values are where, handling futures correctly. The simple fragment of code above actually translates into something quite complex. The complexity, however, is completely hidden from the user.

2.3.2. Ordered and Unordered Replicated Assets

As previously noted, concurrency is further achieved by replicating assets. When using replicas, the synchronization requirements of f-calls can be relaxed by making the replicas unordered in the meta-program. This means that if a future from one call to the asset is referenced but not yet available, the first corresponding future returned by another replica of the asset will be used in its place.

Again consider the code segment above in Section 2.3.1. Assume `AssetA` is an asset that simply returns the value of its argument. If `AssetA` is replicated 5 times and made ordered, the default, then the values in the array `x` would be precisely {0, 1, 2, 3, 4}, as normally expected in a sequential program. On the other hand, if `AssetA` is made unordered, then the values in the array `x` might become any permutation of {0, 1, 2, 3, 4}. This does not preserve the semantics of sequential C. However, in certain applications when the order of an asset's return values is irrelevant, using the unordered property for replicated assets can increase concurrency. The unordered option is strictly an efficiency optimization. It must be used with care.

2.3.3. Parameter Passing Macros

Scalar types and structures used as parameter values are passed in a message to a called asset, but the parameter values are not returned in a message to the caller when the asset has completed its task. This is as expected in standard C, where all function arguments are passed by value. With arrays, however, there is the illusion in C that they are being passed by reference since variables denoting arrays are, by convention, simply pointers to the first element of the array. The pointer is still being passed by value, but implicit or explicit de-referencing of that pointer in the called function can access or modify the values in the array of the caller. To achieve this behavior in *Enterprise* requires the user to explicitly indicate the number of elements to pass. The system then sends precisely that number of data elements in a message first to the called asset and then back again to the caller upon completion of the task. But this message traffic can become unnecessarily expensive when not actually needed in both directions.

To specify the direction of parameter passing for arrays or other pointers, *Enterprise* provides three macros, employed, for example, in a manner similar to those used in the distributed programming language Concert/C [AR94].

- `IN()` specifies that the array gets packed into a message by the caller and sent to the called asset, but no values are returned.
- `OUT()` specifies that no initial values are sent to the called asset, but that the array values are packed into a message to be sent to the caller by the called asset when it returns.
- `INOUT()` specifies that the array is passed in a message in both directions.

Consider the following code in `AssetA`:

```
int  data[100], a;

AssetB(&data[40], INOUT(20) );
/*   More code   */

a = data[50];
```

The single parameter used with these macros is the number of elements of the array to be passed. In this case, 20 integers from `data[40]` to `data[59]` in `AssetA` will be packed in a message and sent to `AssetB`. When `AssetB` returns, 20 integers from the array in `AssetB`'s parameter definition will be packed into a message and sent back to `AssetA`. Meanwhile, `AssetA` continues to execute until the statement that references `data[50]`. It then looks for the future and blocks if the message from `AssetB` has not yet arrived. When `AssetA` finally receives the message from `AssetB`, the values in the

message are used to overwrite `data[40]` to `data[59]`. Execution of the code in `AssetA` then resumes.

If only a size parameter is used without one of these macros, *Enterprise* uses `IN()` as the default. This incurs less overhead but does not result in the same behavior as a sequential function call. When `OUT()` or `INOUT()` are used, the call becomes an f-call, futures are generated, and lazy synchronization is effected. Purely asynchronous calls can have only `IN()` parameters and no explicit return value.

2.3.4. Programming Restrictions

As we have seen, codable assets are accessed only via a call to the name of the asset together with its specified parameter list. This takes the form of a standard C function call in the user's code. However, the occurrence of these asset calls must be compatible with the meta-program design. Assets cannot be called from just anywhere in the program. There is also a restriction on aliasing return values from asset calls. Aliased return values are not recognized as futures. An asset accessing an aliased return value will not block as expected to wait for the value to actually return. Also, while asset calls may contain parameters that are pointers to simple or complex types, they may not contain pointers to pointers or to structures containing pointers. If a pointer is used as a parameter in an asset call, it must be followed immediately by a size parameter indicating the number of elements being passed, with an optional macro to specify the direction of message passing. This requirement, unnecessary in sequential programming where all pointers refer to a single memory space, allows the user to pass, in either or both directions, dynamically allocated arrays, whose size is determined at runtime, and variable portions of arrays. *Enterprise* research is currently involved with the problems of transparently allowing global variables, recognizing aliased return values, and providing a mechanism for automatically marshaling self-referential structures like linked lists into message streams.

The module functioning as the first asset in a program must have as its parameters `argc` and `argv`, declared respectively as `int` and `char**`, as is typical of a C program's main function. None of the *Enterprise* assets or their internal functions, however, can be called `main`.

2.4. The *Enterprise* User Interface

The *Enterprise* object-oriented user interface [Lob93] is written in Smalltalk (ParcPlace VisualWorks 1.0). It requires a licensed copy of this Smalltalk interpreter to run. A simplified X-Windows interface is also available for designing, compiling, and running *Enterprise*. However, the animation facilities and debugging facilities presented in

this thesis are only implemented in the Smalltalk interface. A more complete X interface is under development.

The user interface provides an *Enterprise* window with three views on an *Enterprise* program: the Design View, the Animation View, and the Replay View (described in Chapters 4 and 5). The interface starts up in the Design View. This is where *Enterprise* programs are designed, coded, compiled, executed, saved, loaded, and deleted. The Animation View is used for post-mortem analysis of an *Enterprise* run based upon the recorded event file. The Replay View provides event animation coupled with an execution replay facility under the control of a breakpointing mechanism.

User control is achieved primarily through context-sensitive pop-up menus which appear in the *Enterprise* window when the user presses the <operate>[2] button on the mouse with the cursor in the window. The menus vary with the component being pointed to and the state of the model. Dialog boxes appear with some of the menu selections.

2.4.1. The Design View

The Design View[3] opens with a Program Pane having a single Enterprise asset which contains a single Individual. The user can either load a previously constructed and saved program or build a new program from scratch by using the context-sensitive menus associated with assets to expand them, convert simple assets to composite types, add assets, replicate assets, and then associate names and code with the resulting codable assets. A separate Service Pane can be opened in this view above the Program Pane for adding Service assets. These Services are shown separately since they are accessible by all the assets in the Program Pane. In this way arbitrarily complex programs consisting of nested composite assets can be created, as previously demonstrated in Figure 2.1.

---

[2]In Smalltalk parlance, the left mouse button is the <select> button, the middle button is the <operate> button, and the right button is the <window> button. Elements in the interface are selected using the <select> button. Context-sensitive pop-up menus associated with a window component or view are displayed and selected with the <operate> button. The <window> button brings up a menu to control window operations such as resizing and closing.

[3]This description of the Design and Animation Views is intended to be a summary of the functionality and appearance of the interface, rather than a detailed users' guide. Greater detail can be found in [Lob93] and the Enterprise Users' Guide, which can be obtained through the World Wide Web (WWW) at http://web.cs.ualberta.ca/labs/enterprise.html or *ftp*'ed from the directory pub/Enterprise at ftp.cs.ualberta.ca.

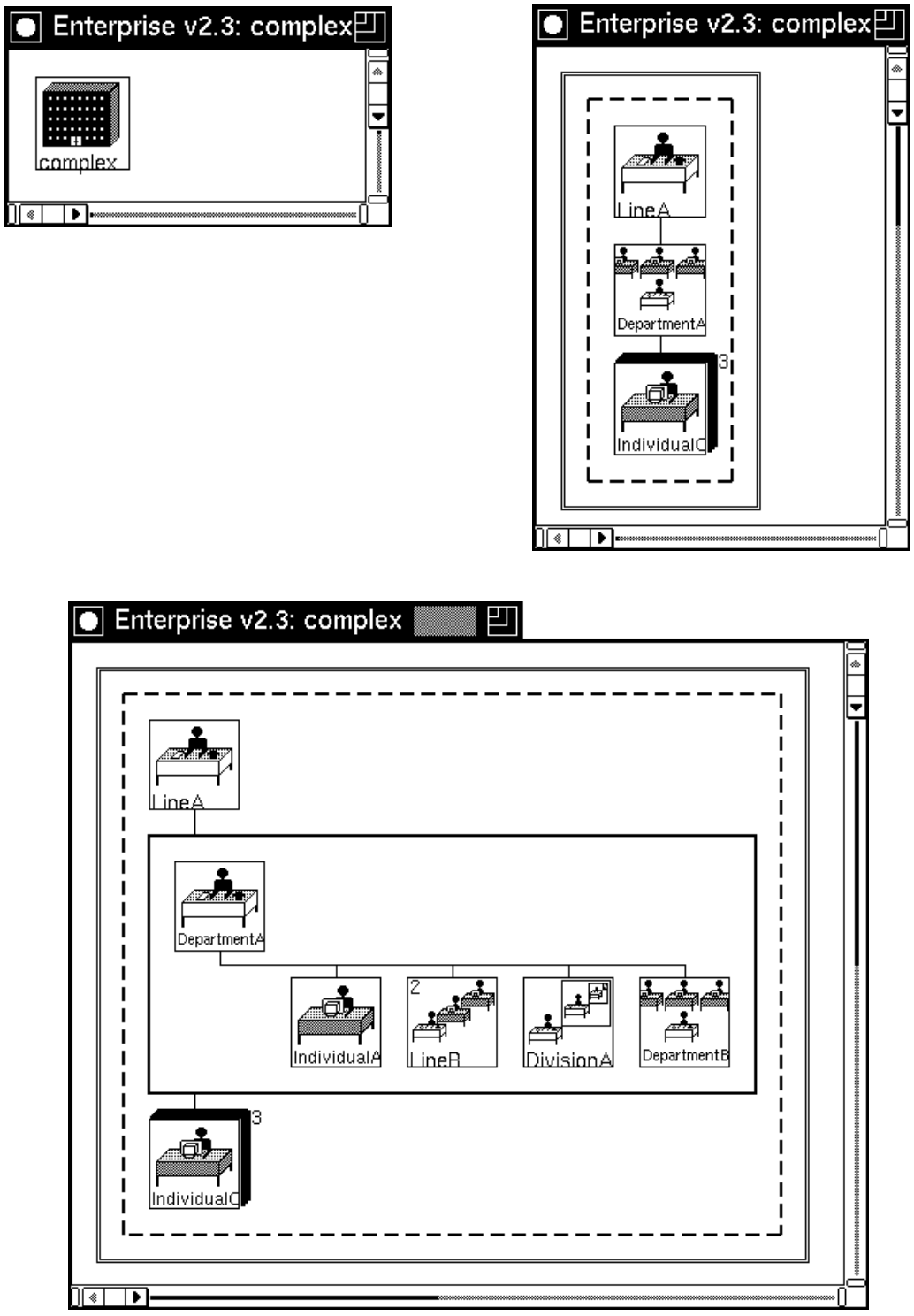


Figure 2.2. A complex *Enterprise* program progressively expanded.

Such a program can be viewed at different levels of complexity by selectively expanding or collapsing composite assets and hiding the Services. Figure 2.2 shows this same complex program with Services hidden and various levels of assets expanded. When a composite asset is collapsed, it is represented by the icon associated with that particular asset type and bearing the same name as the Receptionist. When the composite is expanded, the composite's icon is replaced by a distinctive border—a double width solid line for a Department, a double width dashed line for a Line, a double width shaded line for a Division. The border contains the component assets in a distinctive configuration—a vertical configuration for a Line and Division, a horizontal configuration of components displayed offset below the Receptionist for a Department.

At the other extreme, Figure 2.3 shows the meta-program for the *Enterprise* equivalent of a "Hello world" program called **`CubeSquare`**. **`CubeSquare`** performs the trivial task of summing all the cubes and squares of numbers from 0 to 9. It consists of a Department, `CubeSquare`, with two Individuals, `Square` and `Cube`, each of which is replicated two times. The entire code for these assets is listed in Figure 2.4. `Square` simply returns the square of its argument; `Cube` returns the cube. `CubeSquare` calls `Square` and `Cube` in a loop and then sums these results in another loop. Since the summation operation is both commutative and associative, it makes no difference in what order the results are summed. Therefore, `Cube` and `Square` could both be designated as unordered assets.

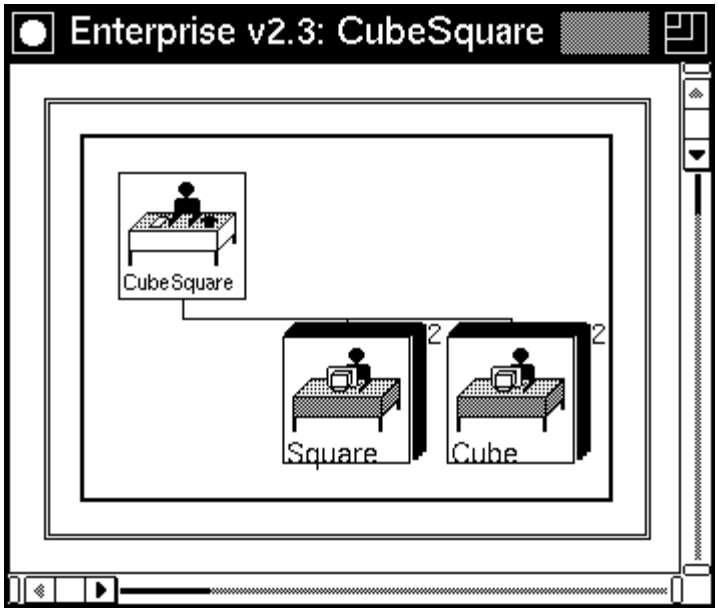


Figure 2.3. The meta-program for `CubeSquare`.

```
CubeSquare( argc, argv )
int argc;
char ** argv;
{
      int i, sq, cu;
      int a[ 10 ], b[ 10 ];

      for( i = 0; i < 10; i++ )
      {
            a[ i ] = Square( i );
            b[ i ] = Cube( i );
      }

      sq = cu =0;
      for( i = 0; i < 10; i++ )
      {
            sq += a[ i ];
            cu += b[ i ];
      }
      printf( "sum of squares %d\n", sq );
      printf( "sum of cubes   %d\n", cu );
}
```

```
int Square( i )
int i;
{
      return( i * i );
}
```

```
int Cube( i )
int i;
{
      return( i * i * i );
}
```

Figure 2.4. The code for `CubeSquare`.

Naturally, this program incurs communication and process management overheads which result in a slow-down compared with a sequential version of the program. If `Cube` and `Square` were replaced by computationally intensive operations, the *Enterprise* version of this program would indeed deliver speedups. As simple as it is, **`CubeSquare`** is sufficiently complex to illustrate most of the features of an *Enterprise* program. Constructing the meta-program for **`CubeSquare`** requires no more than one minute of selecting menu entries and entering names for the assets. The coding is also done in the Design View interface using a Smalltalk text editor, or any editor of the user's choice.

The replication factor and ordering property of an asset are controlled with a Replication Dialog Box (Figure 2.5). By clicking on the up or down arrows, or by entering a new number in the field, the user can change the minimum and maximum replication factors. The maximum is automatically kept greater or equal to the minimum. The asset can be made unordered by clicking on the check box.

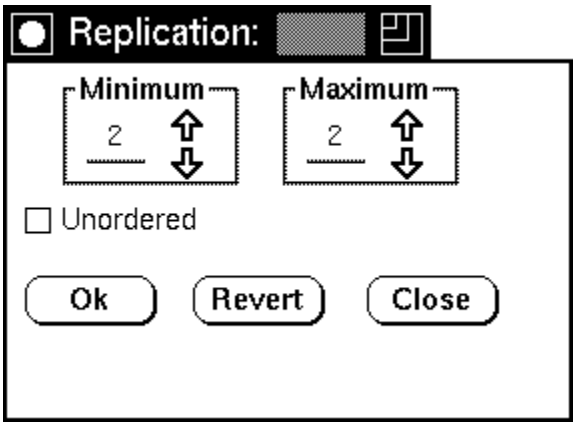


Figure 2.5. Replication Dialog Box.

Assets can be compiled individually, in composite groups, or all together as a program to create an executable *Enterprise* program using a Compile Dialog Box (Figure 2.6). The Compile Dialog Box also allows the user to select the communication manager, set options, specify command line parameters, and link external libraries. Output from the compilation is visible in the top pane.

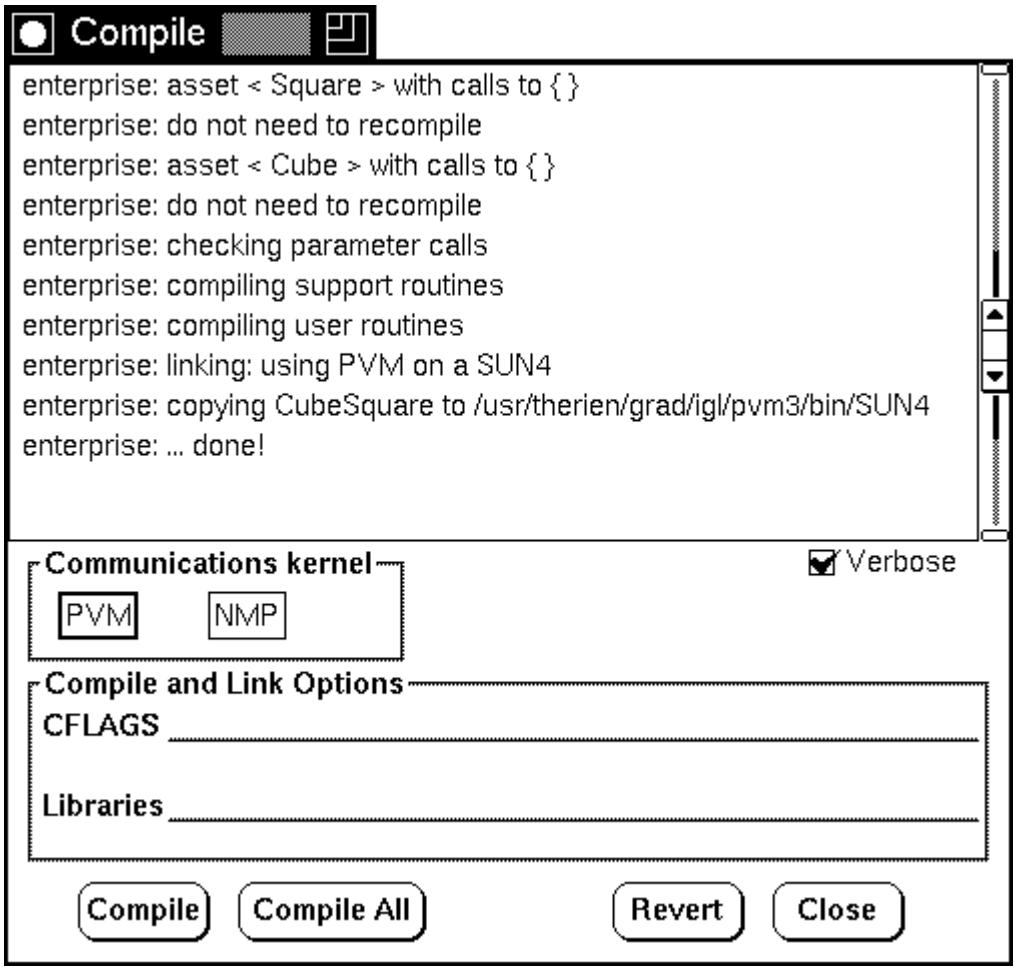


Figure 2.6. Compile Dialog Box.

A Run Dialog Box (Figure 2.7) allows the user to run the compiled program. Here the user specifies the names and architectures of the machines to be used for the run. Machine names preceded by a hash symbol (#) in the Machine File list are "temporarily" excluded by the user for the run. They can easily be included again with a simple menu selection. Output from the run script and the *Enterprise* executive can be viewed in the upper pane. The Run Dialog Box also has a check box for turning event logging on in order to generate an event file for use in the Animation and Replay Views. At any time, the user can abort the program from the Run Dialog Box.

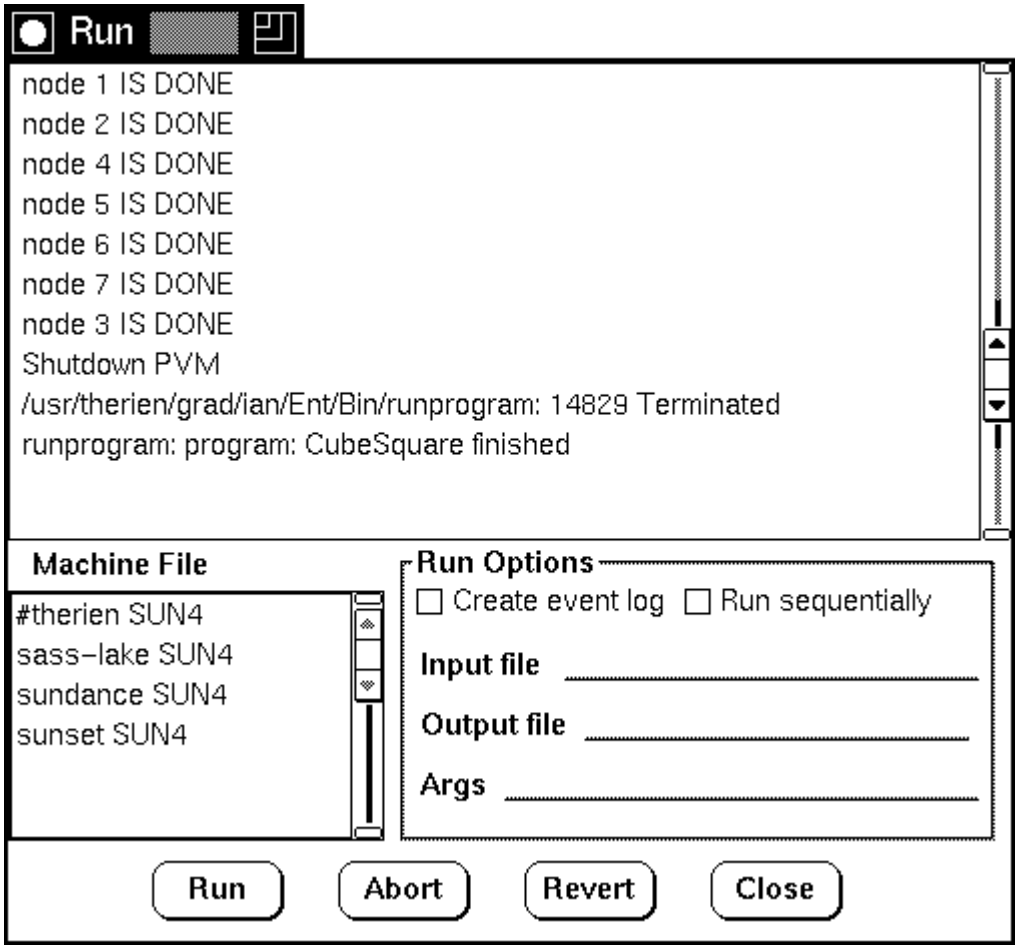


Figure 2.7. Run Dialog Box.

An Asset Options Dialog Box (Figure 2.8), available for each asset, allows the user to finely tune the specification of which machines a particular asset is or is not to run on. Here, the user can also choose to open runtime output and error windows for an asset. As well, debug or optimize options can be set for the asset's compilation.

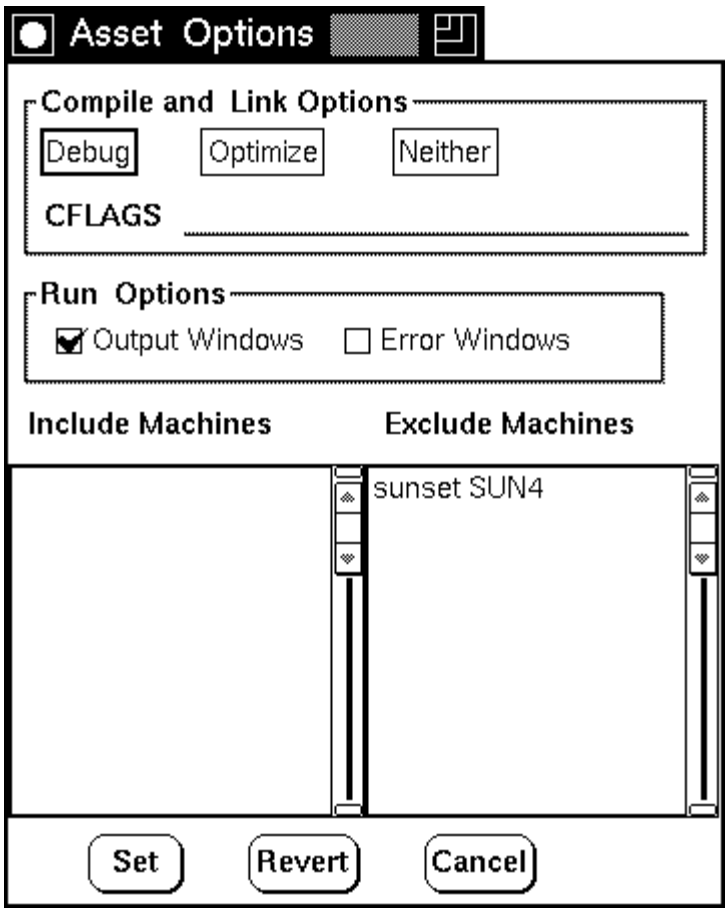


Figure 2.8. Asset Options Dialog Box.

2.4.2. Animation View

The Animation View provides a tool for the post-mortem analysis of a program run based upon the generated event file. The user can simulate a particular execution, without actually re-running the program. This view can be used to analyze the performance characteristics of a program and to detect various programming errors. By viewing the animation, the user can dynamically observe the degree of parallelism, the relative states of processes, the buildup of messages in message queues, and the values of captured parameters in the messages. Figure 2.9 shows the program **`CubeSquare`** animating.

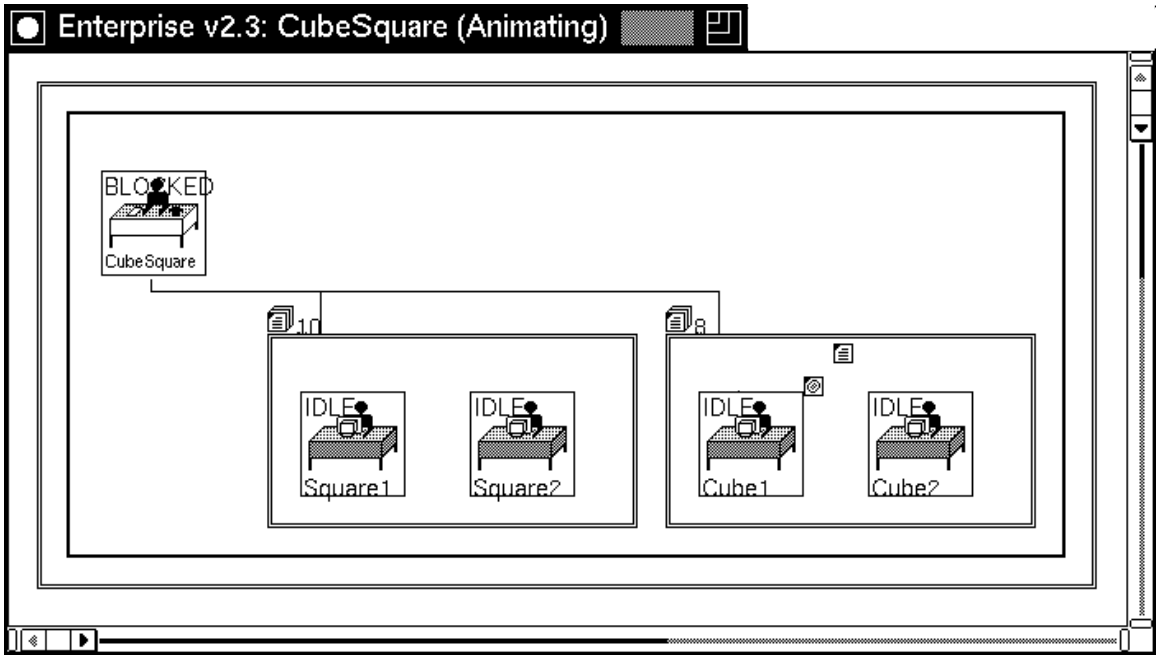


Figure 2.9. `CubeSquare` animating.

After an event file has been logged during a program run, the user can switch to the Animation View. Currently, the Animation View displays the program's assets in their fully expanded form, also expanding all replicas. Each asset is labeled with the name of the state that it is in—IDLE, BUSY, BLOCKED or DEAD. If a color monitor is used, the assets are also colored according to their state—yellow, green, red, or gray, respectively. The asset name for a replica is slightly modified by the addition of a consecutive number, starting with 1, to its base name, so that every replica is uniquely named.

Each asset, in addition, has two message queues: an input queue at the left of its top edge for call messages and a reply queue at the middle of its right edge for return messages. If the queues are empty, nothing is displayed. During animation, messages move along paths on the screen between assets and enter the message queues, which are then visibly displayed by a message icon and a number designating the number of messages in the queue.

At any time, the user can stop the animation and open a window with a Message Queue List (Figure 2.10) for any of the visible message queues by using the menu associated with the message queue icon. Selecting a message from the list in the top pane of the window causes the values of the message's captured parameters to be displayed in the bottom pane. Messages in the list are briefly identified by their sender and a tag. In this particular message queue, all the tags happen to be odd-numbered. Messages with even-numbered tags all went to `Cube`. To illustrate how logged arrays are displayed, the

`CubeSquare` code in Figure 2.4 was slightly modified to include an array of floats in the parameter list for `Square`. Here, the user had decided to log `y[1]`, `y[2]`, `y[3]`, and `y[5]` (but not `y[0]` or `y[4]`) when the asset `Square` was called. Changing the parameter logging specification is discussed in Section 4.5.1. These Message Queue Lists can be kept open when animation is resumed. They will update automatically as messages enter and leave the queues.

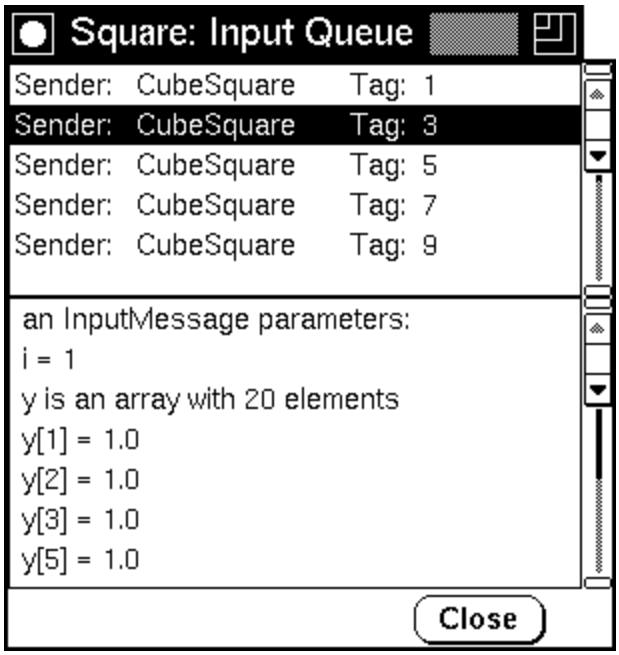


Figure 2.10. Message Queue List.

Replicas of an asset are displayed within a bordered rectangle which represents the entire replicated asset. This rectangle corresponds to a special manager process. The manager receives all the calls to the asset and then looks for an idle replica to respond to the call. Each of these replicated asset managers has its own input queue. But since a manager simply forwards replies immediately back to the caller, no reply queue is shown for the manager. Replies from the replicas animate directly back to the original caller. The name and state of a manager are not displayed.

All assets start off in the idle state. An idle asset is simply waiting to receive an input message. A busy asset is executing user code or actually sending a message, sending a reply, or receiving a reply. A busy asset cannot receive a new input message; it must first complete its current call and enter the idle state. Any input messages for a busy asset accumulate in the input queue. An asset becomes blocked if it tries to access a future that has not yet been returned from another asset. The asset stays blocked until the required future has been obtained from a message in the reply queue. An asset is dead if it has

completed all its work and all of its potential callers are dead. An asset could also become dead due to an error or an abort sent from the interface.

All the events recorded in an event file describe a message-passing event and/or a change of state. The four message-passing events are SentMsg, RcvdMsg, SentReply, and RcvdReply. The events that purely indicate a state change are DoneMsg, Blocked, and Die. The relationships between the four asset states and these seven event types are shown in the state transition diagram in Figure 2.11.

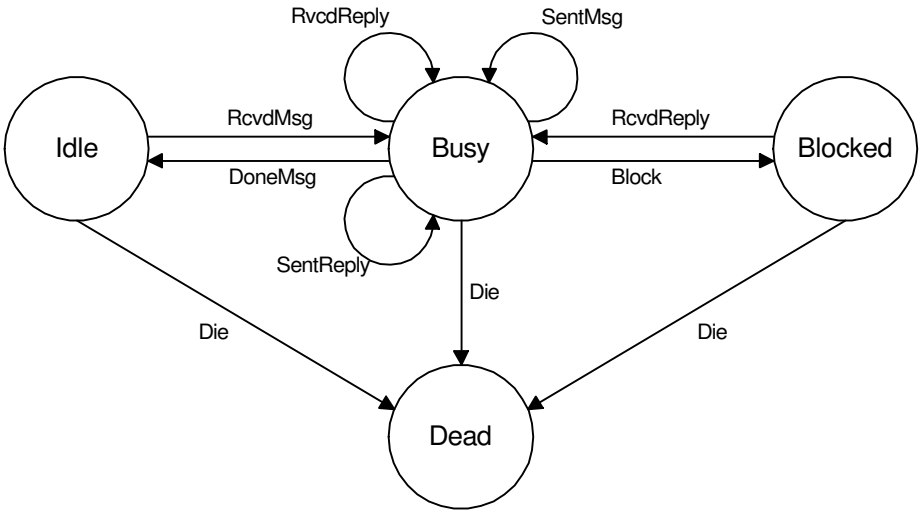


Figure 2.11. State transition diagram for asset states [Lob93].

The animation is controlled by a context-sensitive animation menu. The Department Receptionist `CubeSquare` is blocked waiting for a reply from one of the `Square` or `Cube` assets. The manager for `Square` has 10 messages in its Input Queue. Both of its replicas are idle, since at this moment they have not yet received any work from their manager. `Cube`'s manager has 8 messages in its Input Queue. One message is traveling down from the input queue of `Cube`'s manager to `Cube2`. A reply is traveling up from `Cube1` to `CubeSquare`. Both of `Cube`'s replicas are also idle.

All of the functionality of the Animation View is also found in the Replay View. The animation displayed in the Replay View, however, follows an actual re-execution of the program, not just a display of a previous execution.

2.5. The *Enterprise* Pre-Compiler

The *Enterprise* pre-compiler is based on a modified version of the GNU C compiler. The job of the pre-compiler is basically to detect asset calls and futures and then to trans-

form these into a series of statements to communicate with the asset on a remote machine and to wait for referenced futures to arrive. The result is a modified source file for each pre-compiled asset.

The pre-compiler is currently being rewritten using Sage++ [BBG94], an object-oriented tool that provides classes for manipulating the parse tree, symbol table, and the type table of a program. This high-level view of the user's code allows for greater optimizations, a shorter compilation cycle, and support for other languages (Fortran, C++, Pascal) in future versions of *Enterprise*.

When the user compiles an asset, or the entire program, an automatic Make facility determines which assets need to be recompiled. A forced compile, ignoring the Make facility, is also an option. For each asset being compiled, the pre-compiler is invoked to insert the necessary *Enterprise* code. Then this modified asset source code is sent to a standard compiler to produce an asset object module. As a last step when compiling the program, any special files shared among assets are also compiled into object modules. Then all the resulting object modules are linked with the *Enterprise* system modules, the *Enterprise* system libraries, the communication manager library, and any other user-specified libraries to form an executable program.

2.6. The Communication Manager

Currently, *Enterprise* supports the use of two communication packages to manage inter-process communication across the network. PVM (Parallel Virtual Machine) [BDG91], a widely used public domain communication package, and NMP (Network Multi-Processor) [MBS88], a University of Alberta, Department of Computing Science in-house product. Before compiling the program, the user selects the desired communication manager, which is subsequently linked to the final executable. The *Enterprise* executive uses a small set of generic functions to interface with the communication manager, providing a relatively easy port to other communication packages.

PVM is the default. Communication with NMP tends to be faster than PVM, but reliability problems have occurred with NMP which discourage its inclusion in future releases of *Enterprise*. Moreover, NMP requires root permission to install.

2.7. The *Enterprise* Executive

The *Enterprise* executive, in the general sense of the term, consists of all of the *Enterprise* supplied code, including statements introduced by the pre-compiler into the user-defined code. In a narrower sense, the *Enterprise* executive refers to the *Enterprise* root process, a process that is launched on the host machine when the user starts to run a pro-

gram. This process manages the asset processes on remote machines and serves as a communication channel with the user interface. The general or specific meaning of the term *Enterprise executive* should be clear from context.

A look at a run of **`CubeSquare`** reveals the responsibilities of the executive. The execution of the version of **`CubeSquare`** in Figures 2.3 and 2.4 invokes eight separate processes, each of which could be running on a different machine. Via a shell script, the first process invoked would be the *Enterprise* root process, which runs on the same host machine running the interface. It, in turn, would spawn seven processes across the network under the control of the communication manager on machines selected from user-supplied lists. One process would run the code for the Department Receptionist `CubeSquare`; two the code for the two replicas of `Square`; two the code for the two replicas of `Cube`. Since `Square` and `Cube` are replicated, they each also require an intermediary manager process to allocate the distribution of work to the replicas and forward replies back to `CubeSquare`. The process-processor binding, process spawning, and establishment of communication paths are all handled automatically by the executive process in conjunction with the communication manager, based on the program's graph file and machine file. In addition to the communication paths between communicating assets, the *Enterprise* executive is connected directly to each asset. This direct communication path is used by the assets to report to the executive when they have died (i.e., completed all there tasks) and to send event messages to the executive for it to log into the event file. All eight of these processes, in fact, run executable files compiled from the identical pre-compiled source code. If all the machines being used have the same architecture, there is indeed only one executable file. At startup, each non-root process learns its role from an initialization message sent from the root process. Thereafter, only the code relevant to that process's role gets executed.

Figure 2.12 graphically shows the system configuration for `CubeSquare`.

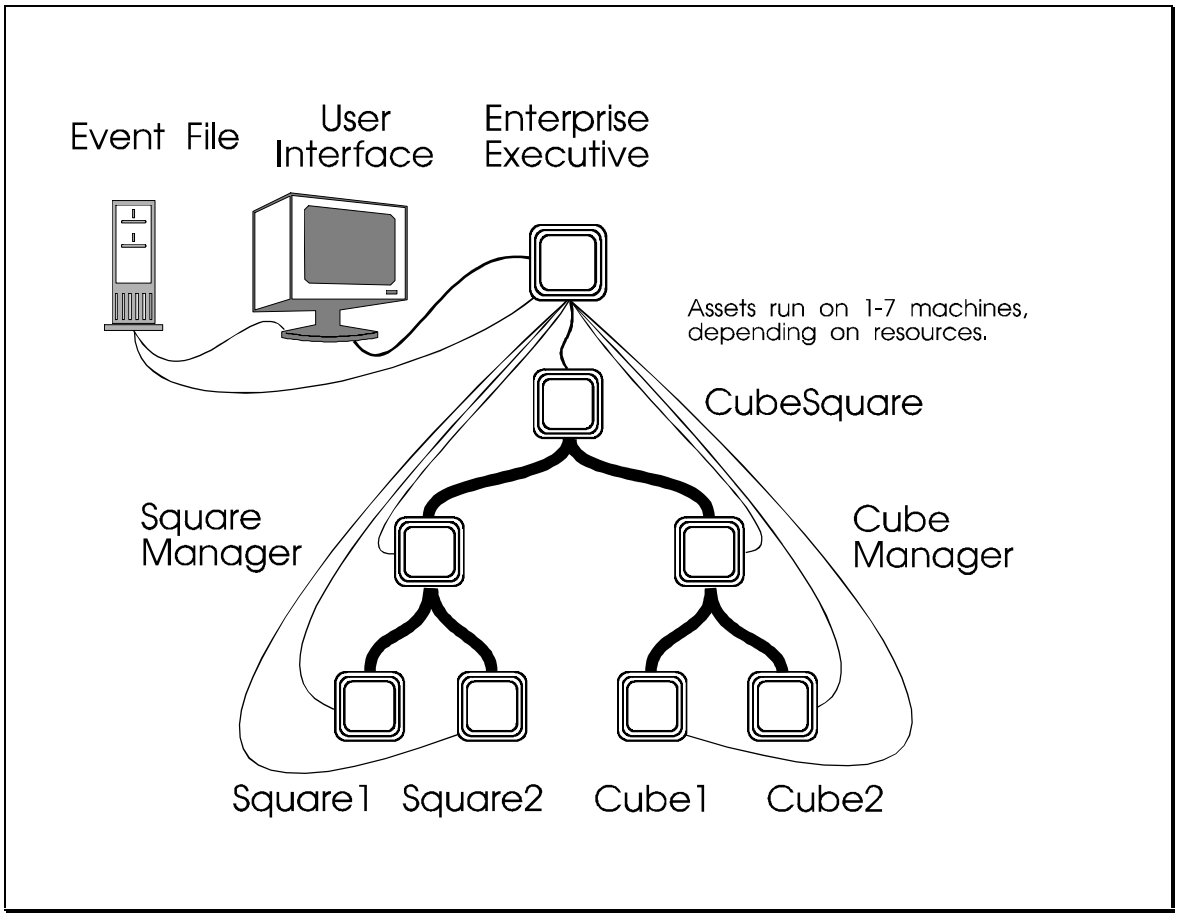


Figure 2.12. CubeSquare process configuration.

Each of the processes goes into a perpetual loop in the *Enterprise* system module where it polls its communication links with connected processes. When it gets a message, it performs the required task, perhaps sending new messages to other processes, waiting for replies, and sending a reply to its caller. When an asset has received a special Die message from each of its potential callers, it knows that there is no more work to be done. It then sends a Die message to all its potential receivers and the root process before it breaks out of its loop and exits. When the root has received a Die message from all the other processes, it breaks out of its loop, performs some cleanup tasks, like sorting the event file, and then exits. During the entire run the root process is also listening for messages from the interface, such as the instruction to abort.

The current *Enterprise* executive is written in C. It is presently being redesigned in an object-oriented paradigm and being re-implemented in C++.

2.8. Related Work

Template-based PPSs for distributed computing are a relatively new development in parallel computing. None of the systems described in the literature have as sophisticated a user interface or such a high-level model as *Enterprise*. Two other recent systems that are at least partially template-based are HeNCE [BDG93] [BDG94] and Parallel Computational Frames (PCF) [FFG94].

2.8.1. HeNCE

HeNCE, Heterogeneous Network Computing Environment, like *Enterprise*, is a graphical parallel programming environment running on top of PVM in a heterogeneous network. It, too, provides facilities for creating, compiling, executing, and debugging parallel programs. But rather than providing high-level parallelization constructs like the Departments, Lines, and Divisions in *Enterprise*, the programmer must explicitly specify the communication graph between sequential nodes. But this basic communication graph is actually a dependency graph where one node must complete execution before a successor node begins computation. Parallelism is achieved by the use of "special-purpose nodes." These are not really nodes at all, but rather graph rewriting rules for the subgraph of nodes encompassed within the special-purpose nodes. The four special node types, loop, conditional, fan, and pipeline, come in pairs that can be nested within each other. A pipeline node is like an *Enterprise* Line; a fan node is like replication in *Enterprise*. Though in some ways more flexible than the *Enterprise* approach, the approach used in HeNCE, user specification of an arbitrary dependency graph, allows for the possibility of cycles of nodes that can result in deadlock. The asset graph approach in *Enterprise*, on the other hand, eliminates the possibility of deadlock.

One of the sophisticated features of HeNCE, not available in *Enterprise*, is a facility for implementing different versions of code for different target architectures. Thus, when a node is actually running on a multi-processor machine, it may be desirable to run a different algorithm than the one used when running on a sequential machine. When the user provides target-specific source code, HeNCE automatically chooses the version needed at the time processors are assigned for a run. HeNCE also utilizes a more sophisticated scheme, a "cost matrix," for mapping nodes to machines. This takes into account the relative cost of running a given subroutine on a given host in light of the other HeNCE processes on that host.

Parameters in HeNCE can be scalars, vectors, or two-dimensional arrays of all base types. Unlike *Enterprise*, structures and pointers to pointer-free types are not supported.

A trace facility, primarily for performance debugging, bears similarities to the Animation View features in *Enterprise*. This visualization tool can be used during program execution as well as for post-mortem analysis. Whereas *Enterprise* allows logged parameters to be examined during animation, HeNCE does not. HeNCE does not provide execution replay based on the trace file, nor does it implement the sort of event-based debugging presented in this thesis.

2.8.2. Parallel Computational Frames—PCF

The PCF system provides a parallel configuration language for manipulating a number of parallelization techniques, or templates, called Parallel Computational Frames. These computational models are implemented in a set of libraries called Arnia. The PCF models now available are Master Slave, Data Parallel, Client Server, and Shared Memory. Generally speaking, these models provide high-level communication primitives between node processes, manage overlapping areas for swapping data, automatically decompose data structures for distribution or recompose structures for collection, and provide automatic load balancing. Pipeline, Producer Consumer, and Recursive models are being planned.

The system is designed and implemented using a layered approach. The foundation of the system is a standard Communication Network layer (such as PVM) which provides the functionalities of inter-process communication. Next, a Message Passing Interface layer, to enhance portability, provides a consistent set of mailbox and communication functions which map on to the lower layer. On top of this is a Runtime Support layer which provides mechanisms and data structures for managing multiple instances of PCF models. Next comes the Tools layer. This layer corresponds to the Arnia libraries for implementing the different PCF models. Finally, the Configuration Language layer provides the means for structuring a user application in terms of intersecting PCF models.

Currently the system supports only the C language, though the Arnia libraries also provide a Fortran interface. The PCF environment automatically generates a makefile to facilitate repeated compilation of an application. The present user interface is textually based. Development of a graphical X-Windows interface is planned. This will provide a more suitable method for defining the application structure.

Chapter 3

The Art and Science of Distributed Debugging

In recent years, the literature on debugging parallel distributed systems has become remarkably voluminous—827 entries in a recent, though arguably incomplete, bibliographic compilation [PU91] [PN93]. Conferences have been devoted to this topic. The researchers in this field have explored many factors which contribute to the design and implementation of a diverse assortment of debugging tools worthy of a book-length survey. Of necessity, this thesis will merely touch upon the most essential issues and a few of the illustrative accomplishments in this "hot" area of computing science research.

Many of the papers on distributed debugging begin with the message, explicit or implicit, that while debugging sequential programs might be characterized as a *difficult art*, debugging parallel programs is a decidedly *painful chore*. The cause of this drudgery is a combination of the special problems encountered in distributed computing and the lack of tools and methodologies for coping with these inherent difficulties. What, then, makes the task of debugging programs in a distributed environment so difficult compared to the sequential situation?

3.1. The Difficulties of Distributed Debugging

Sequential programs are debugged with a well-established set of techniques and tools. The most primitive, but often effective, technique commonly used by programmers is the judicious use of output probes—print statements inserted directly into the program code. No special tools are required. But for compiled languages, the program, or module, must be recompiled, relinked, and rerun with each addition of probes—a potentially time-consuming task.

Symbolic debuggers, which require symbol table information from the executable file, greatly facilitate the sequential debugging process by allowing the programmer to dynamically insert breakpoints to suspend normal execution, examine (and perhaps modify) the program state, and control later execution by stepping through the program. Tracing, which is equivalent to setting breakpoints that automatically record some

information and then continue execution, can also be an effective tool for revealing the source of an error.

These basic debugging techniques are at the heart of cyclic debugging, the classical approach to debugging sequential programs. A programmer engaged in cyclic debugging hypothesizes about the location of the cause of a failure in program execution and then repeatedly executes the program, examining its state at different points in order to refine his hypothesis and narrow down the fault. This cyclical process is dependent on the deterministic behavior of most sequential programs. Given the same input, a typical sequential program will repeat itself exactly, every time it is run.

Unfortunately, this cyclical approach frequently fails for parallel program debugging since parallel programs are typically non-deterministic, that is, different execution paths can result each time the program is run. The different possible execution paths are caused by *races*. Races occur when two or more autonomous processes execute in parallel. If, for instance, a parent process is waiting for a reply from any of several child processes working on different data, it may receive the reply from a different child process on different executions in a random and unpredictable way, totally beyond the programmer's control. The particular resolution of a race may be a function of network traffic, CPU loads on each processor, or non-deterministic communication protocols. Clusters of frequently communicating processes might never, in practice, repeat an identical execution path. The processes themselves can be dynamic, with numerous process creations and terminations, complicating even further the identification of a delinquent process. A program may run flawlessly most of the time but fail intermittently, for no apparent reason. A particular execution path that manifests a fatal error may occur once and not appear again even after millions of trials. Thus, a programmer may never be able to recreate a behavior of low probability. In and of itself, non-determinism is not the cause of program errors, but rather a complicating factor which provides greater opportunity for errors while at the same time obscuring their true cause. In some parallel programs, non-determinism is intended and desirable. Theoretical issues regarding non-determinism are discussed in [DF93].

The cyclic debugging techniques of sequential debugging are also extremely intrusive in distributed systems. This intrusion into the normal execution dynamics can itself affect the program results. Any attempt to observe the behavior of a distributed system may change the behavior of that system. This phenomenon has been referred to as the "Probe Effect," the "Heisenberg Uncertainty Principle" applied to software [MH89], or, to be punny, the "Heisen*bug* Uncertainty Principle." Even a frequently occurring error could be difficult to examine due to the masking effect of debugging statements. For instance, a print statement introduced into the code could modify a critical race in a way that dramatically lowers the probability of some erroneous behavior occurring.

Breakpoints can induce an even greater probe effect. If a breakpoint is set in just one process, the rest of the program processes might keep running until normal termination of the program. Even if all processes are stopped at a breakpoint in one process, they will not be stopped instantaneously due to the network communication delays. Any timeouts in the stopped processes will further complicate the situation. An instantaneous snapshot of the global state of a program at a breakpoint is likely an impossibility. Since there is no global clock in a distributed system, the perfect synchronization of local clocks is fraught with problems. The precise total order of events occurring at physically distant autonomous processors is difficult or impossible to determine. A partial order of events is, as we will see later, the important relationship among events. Continuing execution after a breakpoint involves the same synchronization problems. Even the activity of stepping through a program must be redefined.

Communication delays and limited bandwidth can also render impractical the remote collection of debugging information and the central manipulation of the states of a cluster of processes. Significant communication delays, particularly in a geographically large distributed system, can also exacerbate the phenomenon of error latency existing in sequential programs. An error can propagate widely before it causes an observable failure. In a domino effect, the propagated error can generate a burst of failures that obscure the origin of the error. Even a hardware failure may not be obvious in a distributed system, though it might cause mysterious program behavior.

An additional problem in distributed programs is the shear complexity involved. Distributed programs tend to be large in both code size and execution time. When this intrinsic degree of complexity is compounded by the non-determinacy of multiple processors, the resulting state space can grow exponentially. Moreover, the capability of the human brain to conceptually maintain an understanding of more than a small number of concurrent events is quite limited. The pertinent information regarding an error must be extracted from a massive amount of data generated by debugging tools. Organizing and presenting such data in a meaningful and concise format is no trivial exercise.

3.2. A Model for Distributed Debugging

There are, in theory and in practice, a variety of means to overcome the problems inherent in debugging distributed systems. These approaches can be abstracted and analyzed in a number of ways. A comprehensive model for studying the work in this field, thus, fosters a grasp of the big picture without all the clutter of esoteric detail. It serves to move the *art* of distributed debugging to a *science.*

Cheung, Black, and Manning [CBM87] have developed the most analytical framework to date for studying distributed debugging systems. Theirs is a high-level abstraction to promote the study of current research in this field in a more systematic way. They

envision a distributed debugging system (DDS) as the hub of a debugging environment which is partitioned into three domains: the program domain, the human domain, and the distributed domain (Figure 3.1). The program domain is a static description of the distributed program: the source code together with the expected behavior. The human domain is the programmer with all his attributes. The distributed domain is the underlying distributed operating system and the execution behavior of the cluster of process instantiated from the modules in the program domain. The DDS itself is a collection of software modules for facilitating the debugging process. It manages all the interactions among the three domains via the Program, Human, and Distributed interfaces. With an effectively designed DDS, all communication in the debugging environment can take place through these interfaces alone. The programmer, for example, would have no need to bypass the DDS and interact with the program directly. Furthermore, interfaces designed in a domain-independent way make it possible to construct a portable DDS, which can be transferred from one implementation to another with little effort. Lack of portability, indeed, appears to be a serious limitation of the tools and systems for distributed debugging described in the literature. The authors of the model admit, however, that such a portable design is nontrivial.

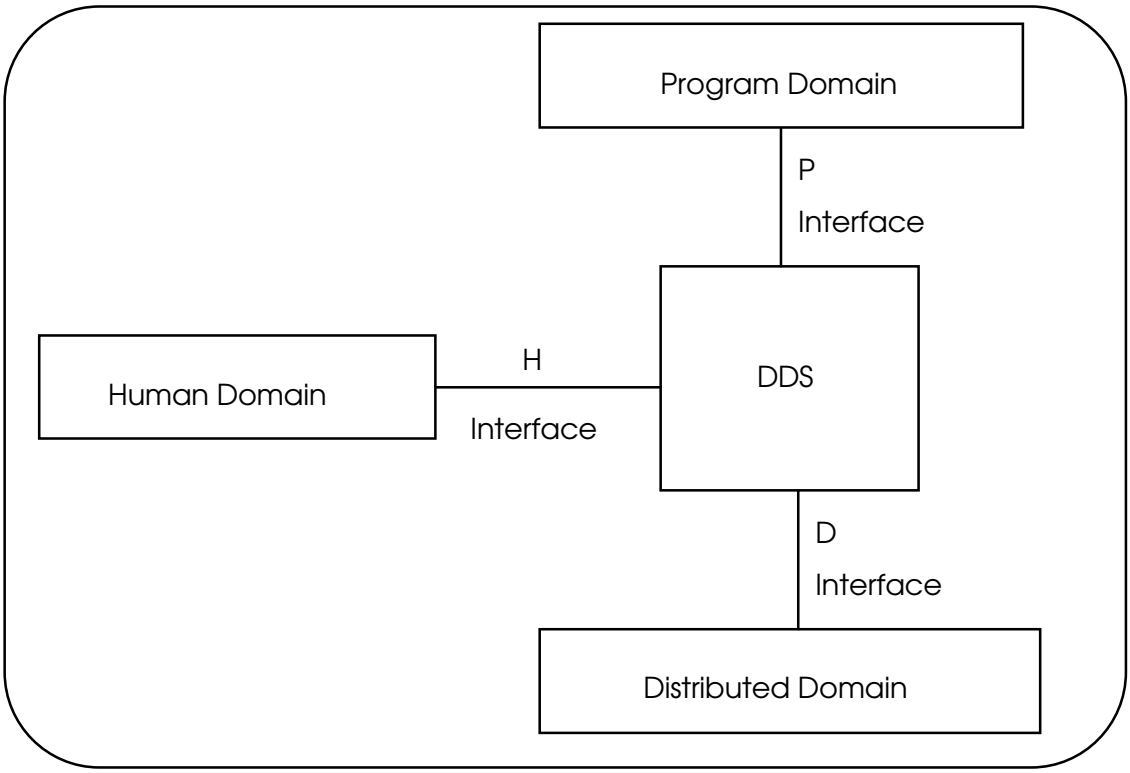


Figure 3.1. Model of a Distributed Debugging Environment [CBM87].

The DDS is decomposed into three problem areas: the distributed design model (DD model), domain interaction, and system support (Figure 3.2). Each of these areas is further

broken down into problem components. The core element of the DDS is the DD model. This conceptual framework for studying the design issues consists of three components: approaches, methodologies, and techniques. Techniques are the most basic component of this model. They are ways in which we can gather pertinent information to further the debugging effort. Output probes, breakpoints, tracing, assertions, controlled execution, replay, rollback, and monitoring are examples of techniques. Methodologies are guidelines to govern the logical progression of debugging activities. They apply systematic reasoning, rather than pure intuition or brute force, to the debugging exercise. Induction, deduction, testing, top-down debugging, bottom-up debugging, and two-phase debugging exemplify different methodologies. The approaches, which provide a general high-level model for the DDS, contain the guidelines for using the techniques and methodologies to capture and manipulate debugging information. The three general approaches identified are the database approach, the behavioral approach, and the AI approach.

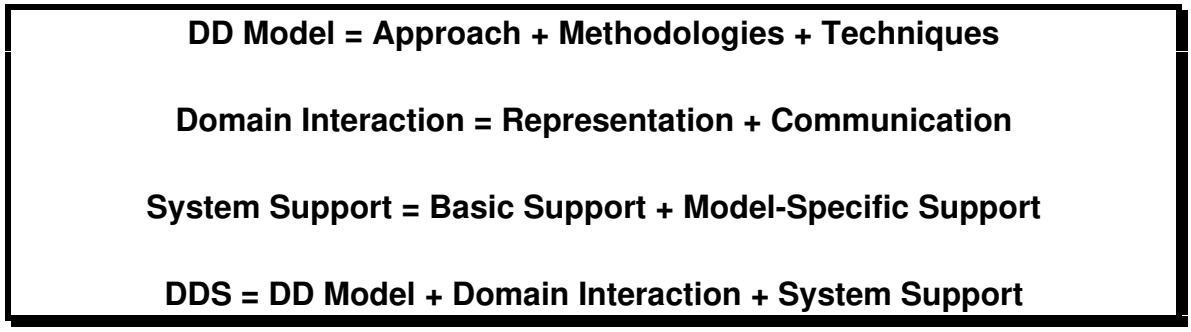


Figure 3.2. Decomposition of a Distributed Debugging System (DDS) [CBM87].

The domain interaction part of the DDS equation consists of two factors: representation and communication. Representation concerns the description of objects or activities in the domains. This description requires an appropriate language to precisely express the essential attributes of these objects and activities. Communication is concerned with the design of the interfaces—the interaction between the domains and the DDS. Protocols are needed to control the flow of information across the interfaces. With improvements in graphical hardware, much of the DDS research effort in recent years has gone into the design of graphical user interfaces—the human-DDS interface. Such graphical representations of information, as opposed to purely textual displays, are well suited to presenting in a meaningful way the enormous quantity of data gathered by distributed debuggers. Interface design was, indeed, a primary focus of the debugger development work in *Enterprise*.

The third problem area in this decomposition of the DDS model is system support—the actual implementation issues. This is further separated into basic support for common requirements of the DDS, such as the domain interfaces, and model-specific support to provide facilities to support a particular DD model. These detailed technical issues are beyond the focus of this survey.

This framework for the study of distributed debugging, though very high-level, is indeed useful for analyzing and conceptually integrating the diverse research that has been done. It must be noted that the issues in this decomposition are not independent. The DD model is the central theme; the design issues of the DD model must eventually be related to the external communication issues and the implementation issues for a particular system. In some cases, however, the questions of user interface design are the matter of central importance, not something simply subservient to a particular technique. Furthermore, the distinctions between conceptual categories, such as techniques, methodologies, and approaches, are not universally agreed upon in the literature. Indeed, much of the most interesting work concerns debugging tools which utilize "techniques" without explicitly embracing any particular "approach" or "methodology."

This model can perhaps be faulted for being an overly ambitious artifice with limited utility when it comes down to the practical problem of designing and implementing a real debugging system. Nevertheless, it does serve the function of providing a conceptual road map for a newcomer to this largely uncharted field, with its myriad issues and entangling details.

3.3. Techniques for Distributed Debugging

It is estimated that 50% to 90% of the effort expended in the life-cycle of software is directed to debugging [CBM87]. As is the case with maintaining human health, prevention can often be more cost effective than a cure. Undoubtedly, there is an important role for such preventative *static* debugging techniques for parallel programs to detect errors without ever executing the program. Design reviews, coding reviews, proofs of correctness, dataflow analysis, and concurrency analysis all have a place in the overall debugging effort. Since no program execution takes place with these static techniques, they completely avoid the troublesome probe effect, which run-time debuggers can only strive to minimize. Despite all the efforts in the prevention of errors, there will still inevitably be run-time errors in programs with high degrees of complexity. This survey only concerns the detection of run-time errors.

Excluding the static analysis systems, parallel debuggers fall into two broad categories: traditional parallel debuggers and event-based parallel debuggers [MH89]. Traditional parallel debuggers are the simplest to implement. They are, actually or functionally, a collection of sequential debuggers, one for each process, controlled with a window manager with one window per process, or by a master window which can accept commands for individual processes or groups of processes. Such debuggers fit the cyclic debugging paradigm, using primarily the techniques of breakpoints and tracing. They are quite susceptible to the probe effect and are ineffective against timing dependent errors. In spite of these shortcomings, they can be extremely effective in finding errors that are not timing related, or where the probe effect does not affect critical races. Most commercial and

public-domain debuggers fall into this category. Examples of graphically based debuggers are Sun Microsystems' *dbxtool* for SunView and the X-Windows debugger *x11ups*. The common textually based debuggers are *dbx* and GNU's *gdb*.

The novel aspects of distributed debugging, however, involve understanding the interaction between processes—the inter-process communication (IPC). In physically distributed systems without shared memory, all communication is via messages. Message sends and message receives are the principal events in IPC. In general, the events of interest are any atomic actions that are visible beyond the scope of a single process. They also include input, output, process creation, and process termination. The event-based debuggers, rather than the traditional debuggers, are the only effective means of understanding the critical IPC. They can provide a more abstract, higher-level view of program behavior, where the events are partially ordered, rather than totally ordered. For two events, A and B, there are three possibilities: A precedes B, B precedes A, or A and B are concurrent. The record of the partial order, indeed, provides more useful information than a record of the total order. A display of the total order can obscure the logical event precedence and possible cause-effect relationships. This record of the partial order can be maintained by attaching timestamps to events [Fid89].

An event-based debugger typically records event information at run-time into an event history which can later be viewed and manipulated by the user. In order to minimize the probe effect, the overhead of recording should be minimal. The question of which information to record depends upon the debugging technique to be employed. In general there are three levels of event-based debuggers and associated techniques, each requiring an increasing amount of detail in the event history and greater system support:

(1) Browsing or monitoring. The event history is manipulated and examined by the user. Graphical displays can be employed to visualize the inter-process behavior. They can include tools for filtering out irrelevant information from potentially enormous records and for clustering related processes and events. A debugger of this nature was recently developed by David Taylor [Tay92] for the Hermes language. This debugger also demonstrates the paramount importance of the design of the graphical user interface (the Human interface in the DDS model) for parallel debugging.

(2) Replay. The debugger uses the recorded information in the event history to control an identical rerun of the processes. Breakpoints can be inserted and messages can be examined in detail. With the non-determinacy of the IPC controlled, each process, assuming no non-deterministic statements or asynchronous interrupts, behaves deterministically. A sequential debugger can be integrated into the system to check the internal state of any process, as in the cyclic debugging paradigm. Some systems allow the programmer to alter the order of events in the history or modify messages to experiment with controlled execution along different paths. One of

the now classic general solutions to the problems of replay is the debugger named Instant Replay [LM87], discussed in greater detail below. This is the level of the event-based debugger developed in this thesis.

(3) Simulation and rollback. If the event history contains all the information necessary to simulate the environment of any single process, then the entire program does not have to be replayed from its beginning. Instead, the single process can be run by itself in a simulated environment under the control of a sequential debugger. This method requires that the complete contents of all messages be recorded, a potentially significant overhead. Rollback, like simulation, avoids the problem of replaying an entire program from the beginning to reach some interesting portion of the execution. With rollback, a snapshot is taken periodically of all the processes, together with an abbreviated history of events. Replay can then begin from a globally consistent state derived from the snapshots which precede the segment of event history which interests the user. The second case study below of DPD (Distributed Program Debugger) demonstrates the use of rollback and replay.

3.4. Incorporating Debugger Methodologies and Approaches

As suggested by the DDS model described above, debuggers designed about a technique without an associated methodology and approach are somewhat incomplete, merely tools and not systems. With such tools alone, the methodology of debugging is left up to the programmer rather than embedded into the system.

The most commonly articulated methodologies in parallel debugging systems are top-down debugging, bottom-up debugging, and two-phase debugging.

Since debugging usually takes place on fully developed systems, most researchers advocate or assume a top-down methodology [BFV86]. The program as a whole is considered first. Erroneous behavior is then localized by progressively more detailed examination of erroneous modules and sub-modules. This methodology eliminates the additional coding effort of writing drivers to debug individual processes.

There is an argument, however, for bottom-up debugging [GGK84]. Since distributed programming is a much more hazardous endeavor than sequential programming, we should choose the most conservative methodology. If individual processes are first made "bug free," then process interface errors can be found more easily. The added effort of bottom-up debugging could save an even greater cost at a later stage in the software lifecycle. These arguments appear more valid when applied to the development and testing of a new program rather than when debugging a fully developed system.

A two-phase debugging methodology [GGK84] is an intrinsic part of most event-based debuggers. During the first phase, the program runs, producing an event history, until some error becomes evident. The events are then examined to localize the error. Phase two involves testing the erroneous process or processes by replay or simulation until the fault is discovered. This sort of two-phase debugging methodology is the logical choice for *Enterprise*, given that it uses event histories.

The general approaches identified for a DD model are the database approach, the behavioral approach, and the AI approach [CBM87]. The AI approach seeks to automate fault localization using expert systems (knowledge-based systems). Though some limited attempts have been made in the realm of sequential and centralized concurrent debugging, this approach is far from any practical realization with distributed debugging.

The database approach is the most common. In fact, any event-based debugger that keeps an event history could be loosely thought of as utilizing a database approach, since the event history is a repository ("database") of program information which can be manipulated in some way by the programmer. A full-fledged database approach will support queries on the values and relationships among the data. Garcia-Molina [GGK84] proposes, but does not implement, SEQUEL queries on a relational database of event histories for his debugger. Powell and Linton [PL83] implemented a database model of debugging for the OMEGA programming system, which uses a rich and powerful set of operations with queries constructed in a "friendly" visual interface. Though designed for sequential programs, the authors considered the problems of distributed debugging as well. In a broad sense, the *Enterprise* debugger uses the database approach without supporting general queries, though conditional breakpoints resemble a limited type of query.

The last approach to consider is the behavioral approach [CBM87]. This also relies on monitoring the execution events. The user must first, however, program the debugger by specifying the target program's expected behavior and defining the action the debugger must take if the program's behavior does or does not conform to the specified behavior. A special behavioral definition language (or event definition language) and action language is used to express these specifications. Although this is unquestionably an additional initial burden on the programmer, it provides a far more powerful mechanism than traditional techniques and approaches. The behavioral definition can be arbitrarily fine-grained, allowing for very selective debugging. Event descriptions can be qualified in a variety of ways and various patterns of events can be tested for. The action statements can allow for such things as invoking a reporting or recording mechanism, halting process execution, or initiating some other debugging technique. Since this event and behavior analysis is done at run-time, there can be serious effects on program performance. The probe effect of these intrusive analyses can be mitigated by combining this approach with a less intrusive event recording mechanism, which in turn can control a replay for the behavioral analysis to be conducted in. Quite a number of researchers have used the behavioral approach. Some of the leading examples are Bates' Event Based Behavioral Abstraction (EBBA)

toolset [Bat89], Bruegge and Hibbard's Path Rules debugger [BH83], and Baiardi's ECSP debugger [BFV86].

Although some of the work in distributed debugging does not fall neatly into some niche in this DDS framework, it is nonetheless a useful conceptual artifice for reviewing the efforts in this field of research.

3.5. Related Debuggers

It is instructive to take a closer look at two debuggers for message-passing systems: Instant Replay and DPD. In many ways they are representative of work in this field.

3.5.1. Instant Replay

Thomas LeBlanc and John Mellor-Crummey [LM87] developed an often-cited general solution, Instant Replay, for reproducing the execution behavior of parallel programs. During program execution, only the relative order and identifying attributes of events are recorded, not the data associated with events. Since the contents of messages are not saved, the event recording takes less time and space than in many other methods. Tests on Gaussian elimination with a 400×400 matrix running on 64 processors indicate a performance overhead of less than 1% during the recording phase. Significantly, there was no additional performance overhead during replay, a phase subject to synchronization overhead in other playback systems. The use of a fully distributed protocol, together with the absence of any global synchronization of events, precludes any centralized bottlenecks. The low performance overhead, indicative of a minimal probe effect, makes this method suitable for use on production runs and on tightly coupled multi-processors, where communication is cheaper and, hence, more frequent. Shared memory machines are as amenable to this method as are distributed architectures.

Instant Replay, in fact, models all process interactions as operations on shared objects. Message passing, in particular, can then be modeled as communication through a shared port. Operations all belong to two classes: read operations which do not change the state of an object, and write operations, that do. Assuming that processes do not contain non-deterministic statements or asynchronous interrupts, it is possible to recreate the sequence of state transitions on all shared objects by recording the sequence of write operations on each object, incrementing version numbers, and similarly recording the version number of each shared object read by a process.

Under this scheme, it is necessary to impose a read-write protocol that ensures a valid serialization of operations on each shared object, thus allowing the partial order of access to each shared object to be recorded in data structures called *process history tapes*.

The authors describe the details of such a protocol, a concurrent-read-exclusive-write (CREW) access protocol. It suffices here to simply remark that these access and recording procedures need to be added to the implementation at an appropriate level (such as compiler code generation). In the replay mode, complementary access procedures can be used to guide execution from the information in the process history tapes.

With Instant Replay, the reproducible program behavior makes cyclic debugging possible. Any statements, most relevantly output statements, can be added to the program during replay, provided they do not affect the sequence of process interactions with shared objects. Any number of replays are possible, allowing the programmer to examine program states or message contents previously overlooked. The principal shortcoming of the system is the necessity of replaying the program each time from the beginning. If the computational requirements are large, the time requirement could become unreasonable. Combining Instant Replay with other techniques, such as checkpointing when a suspicious portion of the execution path can be specified, could add significant utility to the system.

3.5.2. DPD—A Distributed Program Debugger

DPD, developed at the University of Victoria by R. S. Side and G. C. Shoja [SS89] [SS94a], has the capability to stop a program, to roll back to some globally consistent program state preceding a suspected fault, and to replay the original execution. Alternatively, the user can change the behavior of the program during replay by selectively stepping processes and by changing message contents. A graphical interface provides a visualization of the event ordering and communication between processes.

DPD was designed for use in the REM (Remote Execution Manager) environment. REM is a fault-tolerant distributed computing facility which provides the means for processes to create, communicate with, and terminate child processes. Both DPD and REM, written in C, run on top of an unmodified UNIX 4.3 BSD operating system on a local area network of Sun workstations.

DPD creates a User Interface Process (UIP) on the user's local processor and a Remote Debugging Process (RDP) on each of the remote processors. The UIP or the RDP in turn create a standard *dbx* process and an Event Collection Process (ECP) for each process being debugged. The UIP also presents the graphical interface consisting of a Command Window and a Process Display Window. The ECPs detect events in the application processes and send this information to their associated RDP or UIP, which then communicate through REM. The visualization of these events and communication processes then appears in the Process Display Window, similar to Taylor's display without any embellishments. Buttons in the Command Window allow the user to group processes; stop, continue, or step any or all processes; set and clear breakpoints; rerun or quit; and roll back to a selected event.

A checkpointing and rollback algorithm [JZ88], originally developed as a fault-tolerance mechanism for distributed systems, is used. The algorithm is able to find *consistent system states* efficiently from *distributed snapshots* [CL85], unsynchronized views of process states. A system state is consistent if and only if all messages received by all processes have either already been sent in the state of the sending process or can deterministically be sent by that process in the future. If it were possible to take an instantaneous snapshot of an entire system, any state observed under normal execution would always be a consistent system state. But some consistent system states (messages received before being sent!) could never be observed by an omniscient viewer.

Determining the ideal frequency of checkpoints is a problem. Too few checkpoints results in increased time for rollback and replay. Too many checkpoints means excessive recording overhead. One way to ease the problem is initially to set checkpoints infrequently. When a fault is detected and rollback is required, increase the checkpoint frequency so that successive rollbacks will take less time. After a portion of the execution is debugged, reduce the checkpoint frequency again.

The authors identify three areas for improvement in their debugger: (1) the implementation of distributed breakpoints (i.e., when a set of events occurs in two or more processes); (2) less intrusiveness when collecting event and checkpoint information (the ECPs are admittedly a drag on the performance); and (3) enhancements in the user interface.

Chapter 4

The *Enterprise* Replay Facility

4.1. Overview

As indicated in the previous chapter, one of the major difficulties in debugging parallel programs is the element of non-determinism. Although the opportunities for non-determinacy are fortunately reduced in the *Enterprise* model, the phenomenon still occurs. As a consequence of the multiple execution paths that a program can follow, it is possible that a program may run error-free numerous times with the same input data but then suddenly manifest an error. If the execution path for a problematic run is captured in an event file, then a facility can be developed for a controlled re-execution of the program along an identical path. Together with other debugging tools, this sort of replay facility will facilitate the programmer's task of localizing and correcting programming errors, especially those of a non-deterministic nature.

The *Enterprise* replay facility developed for this thesis involved modifications to the *Enterprise* executive and extensions to the user interface. The interface extensions allow the programmer to monitor and control the forced re-execution of a program, while the executive modifications involved setting up special communications with the interface and implanting hooks in the code executed by the assets. The hooks block on all message-passing[1] and Die events, receive instructions from the interface via the executive's root process, and send acknowledgments back to the interface via the root process when the event has been replayed.

4.2. Ensuring the Outcome of Race Conditions

The event file captures sufficient information to guarantee that a program can be re-executed along a completely deterministic path, assuming that there are no non-determi-

---

[1]The four message-passing events are hereafter called Send Message, Receive Message, Send Reply, and Receive Reply. These events are named #sentMsg, #rcvdMsg, #sentReply, and #rcvdReply, respectively, in the event file. In the event file, these events are recorded after the fact, hence the use of the past tense in their names. The perspective in replay, however, suggests the use of the present tense.

nistic constructs in the sequential code of the assets themselves. To ensure repeatability, all input to and output from files accessed by more than one asset, or by a replicated asset, should be handled through a service.

Within the *Enterprise* model, there are several situations where critical race conditions can develop. One such situation involves the Receive Message event. If the receiving asset is connected to only one sender, as in a simple Line or simple Department (Figure 4.1), the behavior of the receiver is deterministic. The communication manager guarantees a first-in-first-out protocol on the pipe connecting the two assets. Moreover, messages cannot get lost, assuming there is no hardware malfunction or system software error. Thus, if `AssetA` sends first Message1 and then Message2 to `AssetB`, `AssetB` will receive Message1 and Message2 in that order.

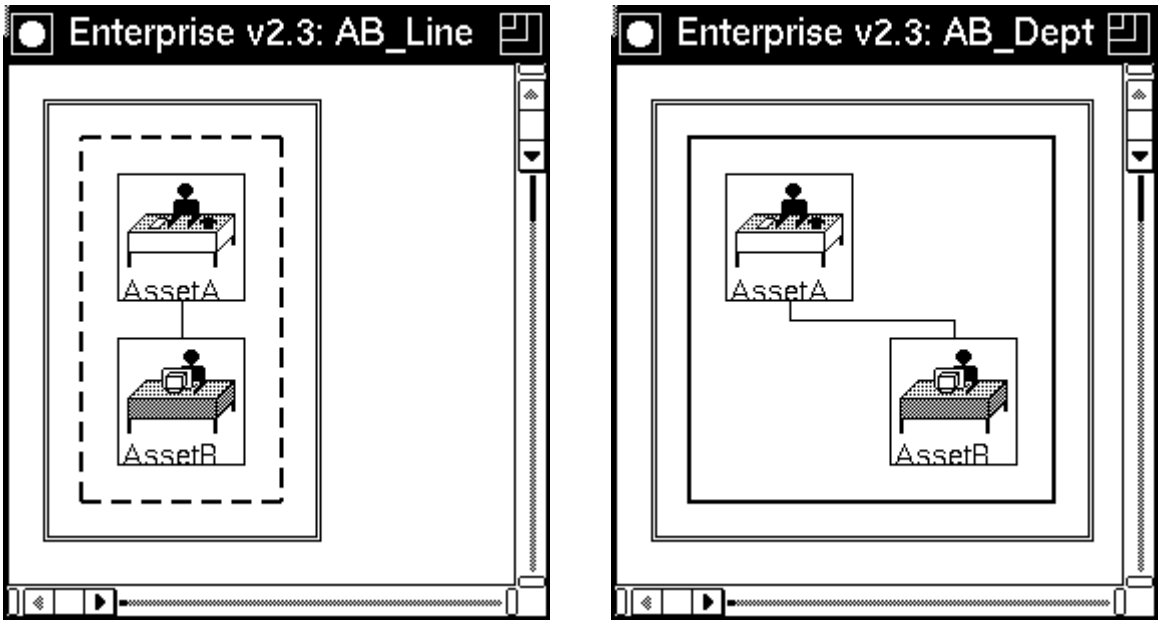


Figure 4.1. A simple Line and simple Department.

However, if the receiving asset is connected to more than one sender, as in a Department embedded in a Line, a race condition can develop. Consider the program **`ManShootsDog`** (Figure 4.2). Suppose `Dog` sends Message1 to `Shoots` and `Man` sends Message2 to `Shoots`. Depending on synchronization factors such as I/O speeds, processor speeds, processor loads, and network traffic, Message1 might arrive at `Shoots` first during one execution, but during another run Message2 might arrive first. Thus, `Shoots` could receive either message first, potentially impacting the end result of the computation.

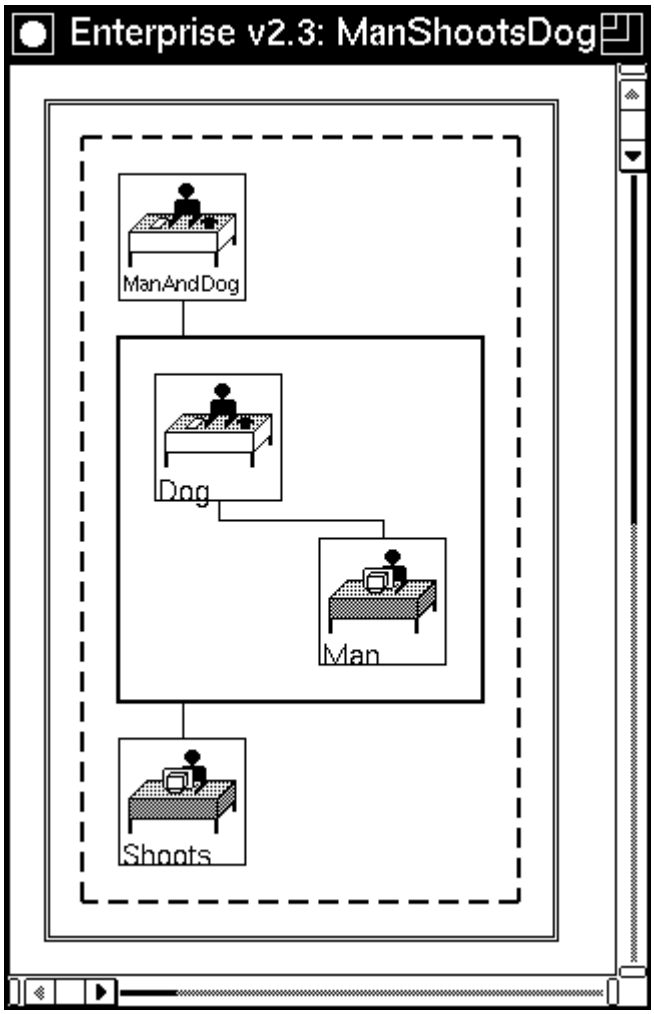


Figure 4.2. Non-determinism #1—Meta-program for `ManShootsDog`.

The code for **`ManShootsDog`** (Figure 4.3), which generates a contrived newspaper headline, illustrates this race condition. If run as a sequential program, the outcome is deterministic: "Man Shoots Dog". But randomizing execution times can yield a more startling result. The randomized execution times for `Dog` and `Man` do not affect the deterministic results of these two assets, but the results of `Shoots` are affected—sometimes "Man Shoots Dog," sometimes "Dog Shoots Man."[2]

[2]This example was inspired by a July 4, 1994 news release by the PAP news agency in Warsaw about a Polish man accidently shot by his dog while he was preparing a target.

```
ManAndDog(argc, argv) /* This Receptionist gets things started */
   int  argc;
   char** argv;
{
  Dog();
}
```

```
Dog()
{
  char* s = "Dog";

  Man();
  WAIT_RANDOM_TIME;       /* An appropriately defined macro */
  Shoots(s, 4);           /* The size of the string s is 4  */
}
```

```
Man()
{
  char* s = "Man";

  WAIT_RANDOM_TIME;       /* A user defined macro */
  Shoots(s, 4);           /* The size of the string s is 4  */
}
```

```
Shoots(s, n)
  char* s;
  int   n;
{
  static int  first_call = TRUE;  /* Assume TRUE & FALSE have */
                                  /* been defined as 1 & 0    */
  if (first_call)
       first_call = FALSE;
  else
       printf(" Shoots ");     /* Not printed on first call */
  printf("%s", s);
}
```

Figure 4.3. Program code for `ManShootsDog`.

Another race condition can develop with a Send Message from a manager to its replicated assets. Consider the meta-program for **`BadMath`** (Figure 4.4). The manager for `Multiply` will send a message to the first idle replica it finds. On one run, the manager might send Message99 to `Multiply1`, while on another run it might go to `Multiply2`. Since all the replicas, `Multiply1` and `Multiply2` execute the same code and the same function calls, it is, perhaps, less likely that a variation in the execution path will produce a different result.

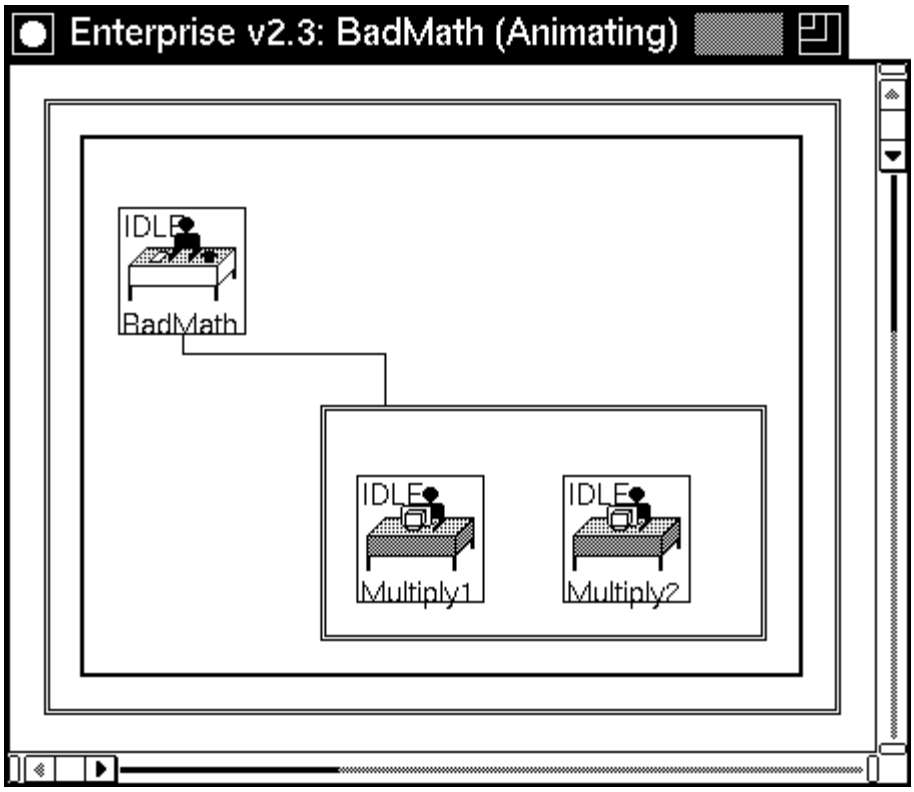


Figure 4.4. Non-determinism #2—Meta-program for `BadMath`.

One scenario is that the replicated asset uses static variables to influence a calculation. The code for **`BadMath`** (Figure 4.5) illustrates this non-determinism, again by randomizing execution times. A controlled replay needs to address such cases.

```
BadMath(argc, argv)
    int  argc;
    char** argv;
{
  int  i, sum, x[5];

  for (i=0; i<5; i++)
    x[i] = Multiply(i+1); /* Non-deterministic with replicas */
  sum = 0;
  for (i=0; i<5; i++)
    sum = sum + x[i];
  printf("%d\n", sum);
}
```

```
Multiply(n)
  int  n;
{
  static int factor = 1; /* Retains value from previous call */

  WAIT_RANDOM_TIME;       /* An appropriately defined macro */
  factor = factor * n;
  return (factor);
}
```

Figure 4.5. Program code for `BadMath`, replicas having static variables.

Yet another race condition presents itself when an asset receives a reply from an unordered replicated asset. When an asset is designated as unordered, the futures associated with return values from that asset can be used by the calling asset in any order. For example, if we want to sum the return values of 10 calls to the asset, we do not have to wait for the result of the first call before we add in the result of the second call, or any other call, since addition is both commutative and associative. Consider a simple example of a program called **`UncertainDivision`** (Figure 4.6). It consists of a Department called `Uncertainty` with an asset called `IdentityAsset`, replicated four times, which simply returns the integer passed as its argument (Figure 4.7). `IdentityAsset` contains a randomized wait so that the calculation times vary unpredictably but do not affect the deterministic result of the function. `Uncertainty` simply has `IdentityAsset` return the numbers from 2 to 5 in a loop and then calls its local function `Divide` in another loop to progressively divide an initial value by the return values, using the standard form of integer division which always truncates the fractional part of the quotient. The result of this cumulative division is the same, 1, regardless of the order in which the return values are divided into the result. `IdentityAsset` can thus be designated as an unordered asset. Any value of `IdentityAsset` which has been returned to `Uncertainty` can be used in the function call `Divide` whenever it is ready. In other words, at the time of the first call to `Divide`, if the future `x[2]` is ready but `x[0]` and `x[1]` are not, `x[0]` will take on the value normally assigned to `x[2]` and be

used in the call. `x[2]` will remain valueless until three calls to `IdentityAsset` have returned.

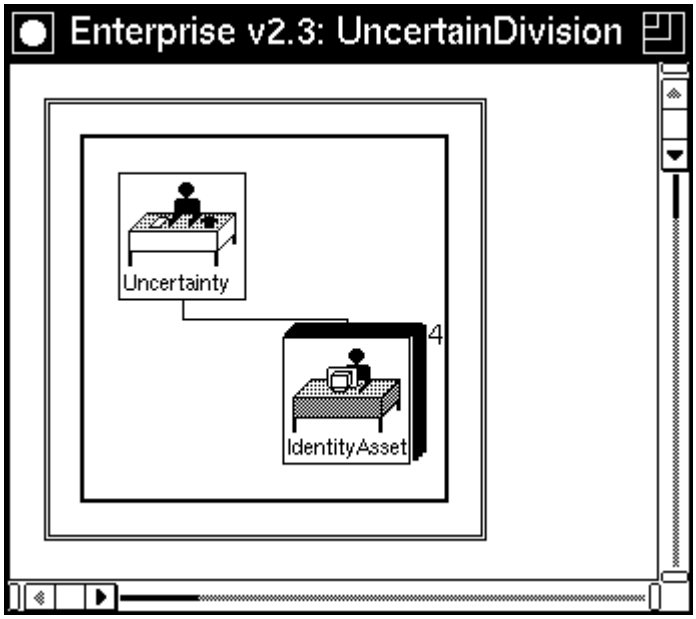


Figure 4.6. Non-determinism #3—Meta-program for `UncertainDivision`.

```
#include <math.h>

Uncertainty()
{
  int x[4], result, i;

  for (i=0; i<4; i++)
       x[i] = IdentityAsset(i+2);
  result = 160;
  for (i=0; i<4; i++)
       result = Divide(result, x[i]);
  printf("%d\n", result);
}

int Divide(dividend, divisor)      /* Standard integer division */
  int dividend, divisor;
{
  return (dividend / divisor);
}
```

```
int IdentityAsset (n)              /* Returns its argument */
  int n;                           /* after a random wait  */
{
  WAIT_RANDOM_TIME;       /* An appropriately defined macro */
  return n;
}
```

Figure 4.7. Program code for `UncertainDivision`.

Now if `Divide` is modified slightly, as shown in the code for `Divide` in Figure 4.8, the results of designating `IdentityAsset` as ordered or unordered are potentially different. The calls to `Divide` now do a different type of integer division using the math library function `nint()`, rounding up or down to the nearest whole number (a fractional part of .5 is rounded "up" to the higher magnitude number). With an initial value for the dividend of 160, the result of the calculation is always 1 if `IdentityAsset` is designated as ordered. But if `IdentityAsset` is unordered, the result is non-deterministic. Some orderings, for example {5, 4, 3, 2}, produce a final result of 2. From this simple example, we can observe the need to enforce the order in which replies are received during replay.

```
int Divide (dividend, divisor) /* Integer division rounding up   */
  int dividend, divisor;       /* or down to the nearest integer */
{
  return (nint ((double) dividend / divisor));
}
```

Figure 4.8. Modified Divide function with rounding to create non-determinism.

These are the only critical race conditions which can occur within the *Enterprise* model.

4.3. The Replay Algorithm

The *Enterprise* user interface provides complete control over the replay of a program. It single steps the program, event by event, according to the information in an event file. The basic algorithm for replaying a single message-passing event is as follows:

Interface: Marshal next replay message.
Send the replay message to the executive.

Executive: Demarshal the message.
Forward control information to the responsible asset.
Poll that asset for an acknowledgment.

Asset: Check for replay mode.
Poll the executive for the replay instructions.
Receive replay instructions.
Check instructions for validity.
Replay event according to the replay instructions.
Send an acknowledgment to the executive .

Executive: Receive the acknowledgment.
Forward the acknowledgment on to the interface.
Wait for another message.

Interface: Receive the acknowledgment.
Begin to process the next event, if appropriate.

The modifications to the executive, therefore, included a startup option to launch all the *Enterprise* processes in replay mode, communications between the executive root process and the interface, additional communications between the root and the asset processes, and hooks for the different controlled event types.

4.4. Modifications to the *Enterprise* Executive

4.4.1. Communications during Replay

When the user selects Start (or Step when the Start option is present) from the main pop-up menu displayed in the Replay View, an *Enterprise* executive process, the root process, is launched by the interface on the host machine. The actual launch mechanism in the interface invokes a UNIX shell script with a replay flag as one of the command line parameters; the script, in turn, starts up the *Enterprise* root process with a replay flag as an argument on the command line. Standard input (*stdin*) to the executive root is a UNIX pipe from the interface. Standard output (*stdout*) and standard error (*stderr*) go through a UNIX pipe to the interface. The root then initializes itself from the program's graph file, spawns the asset processes (with the replay flag set) across the network linked through the communication manager, initializes the spawned processes, and then listens for messages from the interface and the assets.

When the executive root receives a replay message from the interface, it marshals the message for transmission to the responsible asset. After the message is sent to the asset, the executive simply polls the assets waiting for an acknowledgment that the event to be replayed has indeed been executed.

Upon receiving this acknowledgment, the executive forwards it on to *stdout*, a pipe to the user interface. The executive then continues listening to *stdin*, a pipe from the user interface, and the assets for its next instructions.

All of this communication code for the executive root process is localized in the function `ent_process()` and its subsidiary functions in the principal *Enterprise* file, **`enterprise.c`**. The actual sends and receives of replay messages and acknowledgments between the root and asset processes are handled by the same `Send()` and `Receive()` functions, tailored to the selected communication manager, which handle all the other inter-process communication in *Enterprise*.

4.4.2. Message-Passing Events

For the most part, special handling is required in different parts of the *Enterprise* code for each of the four message-passing events: Send Message, Receive Message, Send Reply, and Receive Reply. Moreover, these events are handled differently, in different parts of the code, for managers and non-managers. Thus, there are eight special cases which require consideration.

Three types of message-receiving events for replay, *Receive Message for managers or non-managers* and *Receive Reply for managers*, are captured at the beginning of the main processing loop in `ent_process()`.[3] A validity check is made on the event type, and then the message sender of the replay event is polled until a message is pending.

When a message or reply is ready to be received from the specified sender, each of these three event types is handled in a different specialized code segment. In the case of a manager receiving a message, rather than searching and waiting for an idle replica to which to farm out the work, the manager proceeds immediately to receive the message from the sending asset and sends the executive root the acknowledgment. The manager then executes specialized code to wait for a Send Message replay message from the executive before forwarding the message down to a replica.

In the case of a manager receiving a reply from a replica, the reply is received and an acknowledgment is sent to the executive. Specialized code is then entered to wait for a Send Reply replay message before forwarding the reply back up to the calling asset.

For a non-manager receiving a message, the procedure is similar. The awaiting message is received, an acknowledgment is sent, and the user-defined function for that asset is executed using the parameters contained in the message. If the user-defined function has a non-void return value or OUT parameters, a specialized section of code is executed to wait for a Send Reply replay message.

Three types of message-sending events for replay, *Send Reply for managers or non-managers* and *Send Message for managers*, are handled in the specialized sections of code referred to above. The routine is the same for all of them: if in replay mode, wait for and receive the replay message; check the validity of the message; replay the event; and send an acknowledgment to the executive process. After each of these events a new processing loop is begun.

The *Receive Reply event for non-managers* is somewhat different. The hook for this event is embedded in the function `_ENT_Wait()` in **`enterprise.c`**. The calls to `_ENT_Wait()` are not in the main processing loop of `ent_process()`, but rather were inserted by the *Enterprise* pre-compiler into the user-defined code, using a collection of *Enterprise* stub routines, whenever a future is accessed. In `_ENT_Wait()`, the executive process is polled for a replay message. When it arrives, the replay message is

---

[3]It should be kept in mind that all *Enterprise* processes, the root, Managers, and non-Managers, run binaries built from identical source code. Different portions of the code are executed depending upon the identity of the process.

received, the message is demarshaled, and a validity check is made. Then the designated sender in the Receive Reply event is polled until a message is pending. The message, containing the values of futures, is then received from the designated sender, an acknowledgment is sent to the executive, and the futures are properly queued. Non-determinacy is avoided for values being returned by a manager since the manager must send back the replies from its replicas in precisely the order specified in the event file.

Finally, the *Send Message event for non-managers* is replayed through a hook placed directly into the user-defined code by the *Enterprise* pre-compiler. This is accomplished by stub routines called by the pre-compiler to insert lines of C code directly into the user's code whenever an *Enterprise* asset is called. The hook is implanted using the function `print_body()` in the file **`send_msg.c`**. The same basic procedure is implemented: if in replay mode, wait for a replay message; check the message validity; perform the actual message send; and send an acknowledgment to the executive.

4.4.3. Die Events

A Die event is generated when an asset has performed all its work and its callers have all died. To replay a Die event, the asset simply sets its replay flag to false (thereby turning off replay mode) and proceeds to execute its terminating behaviors: confirming the deaths of its callers and notifying its callees and the executive of its imminent demise.

4.5. Extensions to the *Enterprise* User Interface

4.5.1. The Design View

The Design View contains a facility for specifying which parameters are to be logged in the event file. When an *Enterprise* program is compiled, the pre-compiler generates a file, ***assetName*`.fmt.dflt`**, for each *Enterprise* asset. The syntax for this file in BNF notation is given in Appendix A. This file contains information on the names and types of all the parameters in the parameter list of the asset's entry function and the asset's return value. In addition, the default logging specifications are given to each parameter in the <log spec> field.

The default specification is to log all scalars[4] on asset calls and scalar return values when the asset returns. All vectors of scalars also have their first three and last two values logged on an asset call. The first asset in a program, however, has no logged parameters. The current version of *Enterprise* has, as yet, no facility for logging structures or pointers to data types other than scalars. This deficiency will be addressed in the future.

The user can modify the default settings for parameter logging by selecting Parameter Logging from the pop-up menu for an asset while in the Design View. Such customization can be done only after the asset has been compiled since it requires the information in the ***assetName*.fmt.dflt** file. When the user chooses to edit the logging specifications, a UNIX script, **validate_logging**, analyzes the ***assetName*.fmt.dflt**, created by the pre-compiler, and a file named ***assetName*.fmt**, containing the current logging specifications. If ***assetName*.fmt** does not exist, it is created by copying ***assetName*.fmt.dflt**. If ***assetName*.fmt** is newer than ***assetName*.fmt.dflt**, it is considered to be valid. However, if ***assetName*.fmt.dflt** is newer, then the names and types of all the parameters in the two files are compared. If the names and types are identical, then ***assetName*.fmt** is assumed to be valid and is given a current UNIX file timestamp. On the other hand, if an inconsistency in names or types is detected, a new ***assetName*.fmt** is made by copying ***assetName*.fmt.dflt**. By applying these rules, any parameter logging changes that the user might have made for an asset will not be lost if the asset is recompiled, unless the parameters themselves have been modified.

After the script has run, the interface displays a dialog box containing a list view of the parameters and their logging specifications, based upon ***assetName*.fmt**, together with a specification editing facility. Figure 4.9 illustrates a Parameter Logging Editor for an asset named `Square`, which has two parameters (`i` and `y`)[5] and a return value (`^`). The asset is identified by name in a read-only field at the top of the dialog box. The Logging Specification list view on the left provides the current logging specification for all the parameters and the return value. Each entry in the list view describes the logging specification according to the format specified for <param info> in Appendix A. Some parameters may appear in two lines, one for input logging and another for output. When an entry is selected, pressing the <operate> button in the list view pops up a menu for changing the

---

[4]*Scalars* refers to the fundamental built-in C types: `char` and the `int` types, signed and unsigned, and the floating-point types.

[5]The function `Square` in Figure 2.4 was slightly modified to include an array of integers `y`, for the purpose of illustrating the functionality of the Parameter Logging Editor.

logging specification for the highlighted selection. This context-sensitive menu only shows the relevant menu choices. The possible choices are:

Log on input
Log on output
Do not log

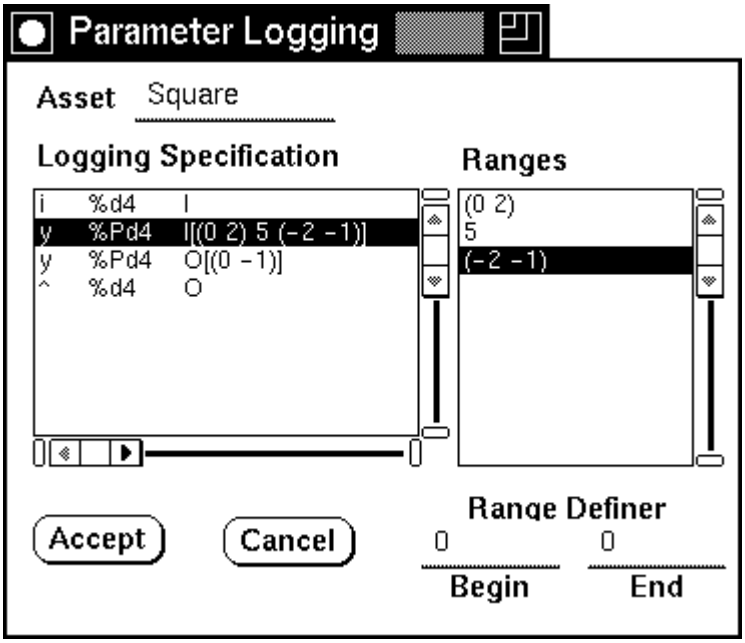


Figure 4.9. Parameter Logging Editor.

These menu choices change the <log code> for the parameter. The <log code> part of the <log spec> field of the <param info> indicates if logging is performed on the parameter when the asset is called (**I**, for ***i****n*), when the asset returns (**O**, for ***o****ut*), or not at all (**X**, for ***ex****clude*). If a parameter is logged on both *in* and *out*, then the parameter must be given two separate specifications. This situation does not occur with the default logging specifications in the **`.fmt.dflt`** files but can occur in the modified **`.fmt`** files. The parameter type is specified in the second field of the <param info>, according to the BNF for <param type> in Appendix A. Scalar parameters are by definition input parameters only, so they can only be logged on input. Return values (`^`) can naturally only be logged on output. Arrays of scalars, designated by a `%P` followed by a `d`, `f`, or `c`, can be logged on input and/or output. As mentioned earlier, more complex structures cannot currently be logged. The <ranges> part of the <log spec> is shown for arrays of scalars being logged. For readability, if <ranges> is empty it is displayed with ellipses as **`[ ... ]`**.

When an array of scalars is selected in the Logging Specification list, as in Figure 4.9, the Ranges list view on the right becomes active and displays the ranges in a list form.

The Range Definer fields in the lower right can then be used to specify a Begin and End point for a new logging range. If the Begin and End points are the same, or if only a Begin point is specified, a single element will be added to the Ranges list. Negative indices reference from the end of an array, -1 being the last element in an array. A context-sensitive pop-up menu can be accessed in the Ranges list. The possible selections are:

Add range
Delete range

Add range performs the function of adding the range specified in the Range Definer to the Ranges list. A notifier appears if the range is illegal or not yet specified. After a range has been added, the Range Definer resets to **0 0**. The added range appears at the end of the list if no entry in the Range list was selected. If an entry was selected, the new range will be added immediately above the selection, in accord with Smalltalk conventions. Delete range removes the selected range from the list. The Logging Specification list automatically updates when changes occur in the Ranges list.

Figure 4.9 shows a parameter `y`, which is an array of integers (`%Pd4`), each consisting of 4 bytes. On input, the first three elements (indexed 0, 1, and 2), the sixth element (indexed 5), and the last two elements (indexed -2 and -1) are to be logged. On output, the entire array (indexed from 0 to -1) is to be logged.

When logging specification changes have been made, the Accept button, initially grayed out, becomes active. When the Accept button is clicked, the new specification is written to ***assetName*.fmt** and the dialog box closes. If the Cancel button is clicked, the dialog box closes without modifying the original logging specification in ***assetName*.fmt**.

The Parameter Logging Editor window can be resized to a larger dimension, thereby proportionately increasing the size of the list views. The window cannot be made smaller.

When the program is run from the Run Box in the Design View with event logging turned on, the script **validate_logging** is run on the ***assetName*.fmt.dflt** and ***assetName*.fmt** files for each of the assets. After *Enterprise* is launched, it then determines which parameters to log by examining the ***assetName*.fmt** files.

It should be stressed that the probe effect of event logging is increased by the additional logging requirements specified for parameters. Keeping this fact in mind, the user should avoid logging unnecessary information. On the other hand, reducing the logging information lessens the control potential of the breakpoint facility and requires the user to rely upon sequential debuggers for examining parameter information. There is no additional probe effect during replay, regardless of the amount of breakpoint analysis being performed, since the replay is fully deterministic.

4.5.2. The Replay View

In addition to the pre-existing Design View and Animation View, a new Replay View has been created to enable controlled replay of a program from an event file. This view, in fact, utilizes nearly everything from the Animation View. Control is achieved by means of a pop-up menu and a breakpointing facility described in the next chapter.

The Replay View is nearly identical in appearance to, though different in functionality from, the Animation View. While the Animation View allows for the simulation of a program's execution, the Replay View provides the means to actually perform an identical re-execution. Replicated assets are fully expanded. Like the Animation View, message passing is animated and message queues can be monitored.

4.5.2.1. Main Menu

The main pop-up menu in the Replay View is activated with the mouse <operate> button while the cursor is anywhere in the view except on an asset icon, message queue icon, or breakpoint icon. The menu choices are dependent upon the current state of the system. The possible choices are as follows:

Start	This option occurs when a replay is not in progress. Selecting Start launches a re-execution of the program according to the event file's sequence of events. Upon launching, the Run Box is raised to the foreground, or created if one does not already exist. The initialization messages for the program and any standard output appear in the text collector view of the Run Box.
Stop	This is the single choice available when replay is running. When selected, the replay and animation pause between discrete events.
Resume	Once a replay has begun but has been stopped by a triggered breakpoint, the Stop option, or after stepping, the choices Resume and Reset replace Start. Resume allows the replay to continue from where it was paused.
Reset	This choice sends an abort message to the *Enterprise* executive, which terminates all the program's active processes. The Replay View is reset to its initial state. Reset and Resume are then replaced as menu choices by Start.

Step	Step can be used from the initial state to start a replay. The program will be launched and then stopped before the first Send Message event is processed. If used when a replay has been stopped (paused), the next event in the event queue will be processed. If this is a message-passing or Die event, the interface will send a replay message to the *Enterprise* executive to execute it.
Set Option	This is identical to the Set Options command in the Animation View. The only difference to point out is that the number of events to process in each animation loop in the Replay View is defaulted to 1, while the default in the Animation View is 5. For the breakpoint facility to exercise full control over replay, it is necessary to limit each animation loop to one replayable event.
Breakpoint Browser	This option will open a Breakpoint Browser, described in the next chapter.

Aside from the main pop-up menu in the Replay View, there are specialized menus for assets and message queues.

4.5.2.2. Asset Menu: Sequential Debuggers

The pop-up menu that appears for an asset has a single option, Sequential Debugger, to attach a sequential debugger. This option appears only after a program has been started and then stopped. Moreover, it is necessary for the program to have been compiled with the debug option on. Sequential debuggers can only be attached to codable assets. Managers, which do not execute any user defined-code, cannot be debugged sequentially using the interface. The debugger is not directly associated with the user's original code, but rather with the code generated by the *Enterprise* pre-compiler containing modifications of the user's code. The intermingled *Enterprise* code and user-defined code is potentially confusing for the novice user. However, with a modest effort the user can effectively utilize the sequential debugger to locate difficult to find errors.

The user can specify a sequential debugger in the environment variable ENTERPRISE_DEBUGGER. If no debugger is specified, *gdb* is the default. The relevant process halts when a debugger is attached. A breakpoint can then be set, say, on the function bearing the asset name. The debugger must be given the command to continue before the process will resume execution.

4.5.2.3. Message Queue Menu: Message Queue Lists

During replay, message passing is fully animated, as in the Animation View. When a message or reply is sent to an asset, it enters an input queue or reply queue represented by a message queue icon displayed adjacent to the receiving asset's icon. Clicking on this message queue icon with the <operate> button, while replay is stopped, brings up a menu with the single choice List messages. Selecting this operation brings up a Message Queue List (Figure 2.10) containing two views: ① a list view of all the messages in the queue labeled by the sender's name and the message tag number; ② a text view to display the values of all the logged parameters in the message selected in the list view.

The display in the text view was significantly modified in this version of the user interface in order to accommodate the new format of the event file (Appendix A). Logged parameter values are listed one per line in a format that identifies the parameter by name and, if necessary, by subscript. When the replay is resumed, the message queue list updates automatically whenever a change occurs in the associated message queue.

Chapter 5

Breakpointing in Replay Mode

5.1. Overview

The *Enterprise* breakpointing facility is accessible in Replay mode. The facility requires the existence of a valid event file, created by executing the current program configuration from a Run Box in the Design mode. As such, breakpointing in *Enterprise* is a post-mortem debugging tool utilized during a deterministic guided replay of a previously logged run.

Breakpoints can be set only at a high level, in terms of message-passing events. These breakpoints can be either unconditional for a particular event type at a particular asset grouping or conditional upon the values of any parameters which have been captured in the event file. When a set breakpoint is triggered, the guided replay stops just *before* the event is executed. It is then a simple operation for the programmer to single step through the event, examine the complete contents of a logged message, or attach a sequential debugger to any process for lower-level debugging.

A largely text-based Breakpoint Browser is coordinated with the graphical representation of the program while it is animating and replaying in the Replay View. This visualization technique is in accord with a core goal of *Enterprise*: to provide an intuitively comprehensible interface to an inherently complex and potentially confusing parallel architecture.

5.2. Event Breakpoints

A breakpoint is defined for an individual codable asset, or for an arbitrary group of replicas of a codable asset. Composite assets, such as Departments, Lines, and Divisions, which have no code directly associated with them, cannot have breakpoints. Thus, a single breakpoint could be associated with two replicas of an asset, but not with two assets having different base names.

There are only four types of message-passing events relevant to breakpointing: a Send Message Event, a Receive Message Event, a Send Reply Event, and a Receive Reply

Event. Every breakpoint is associated with one, and only one, of these event types, but not all event types are meaningful for a particular asset. For example, the first asset in a program cannot receive messages or send replies.

The assets that a given asset communicates with are its collaborators. For a Send Message or Receive Reply event type, an asset's collaborators lie below it in the asset diagram; for a Receive Message or Send Reply event type, the collaborators lie above the asset. A breakpoint must have one or more collaborators associated with it. As was the case with a breakpoint's assets, a breakpoint's collaborators must share the same base name.

Once a breakpoint's assets, event type, and collaborators are defined (as described below), a listing of logged parameter names and types is available from the system. A breakpoint's conditions can be any conjunction of conditional relationships between these parameters and user-supplied constant values. The conditional relationships supported by the system are '==' (equal), '!=' (not equal), '>' (greater than), '<' (less than), '>=' (greater than or equal), and '<=' (less than or equal). For instance, for a logged parameter `y` we could set the two conditions: `y` >= 0 and `y` < 10. The breakpoint would then be encountered when both these conditions were satisfied for the specified assets, event type, and collaborators.

Finally, a breakpoint is not necessarily triggered each time it is encountered. The user can specify the number of encounters before a breakpoint is triggered. The default is to break on the first, and only on the first, encounter. If the user wishes to break on every encounter, an Always Break option can be checked. Thus, if the user wishes to break on the fifth encounter, the number of encounters can be set to 5. If the user desires to break on *every* fifth encounter, both the above settings can be made.

5.3. The Breakpoint Browser

The integrated tool for managing breakpoints is the Breakpoint Browser (Figure 5.1). Its overall design is similar to the Smalltalk System Browser. The top portion of the browser consists of four hierarchically dependent list views for navigating through the breakpoint system. The bottom portion contains the mechanism for actually defining, editing, and accepting breakpoints. The Breakpoint Browser can be enlarged with proportional enlargement of the list views. It cannot be resized smaller.

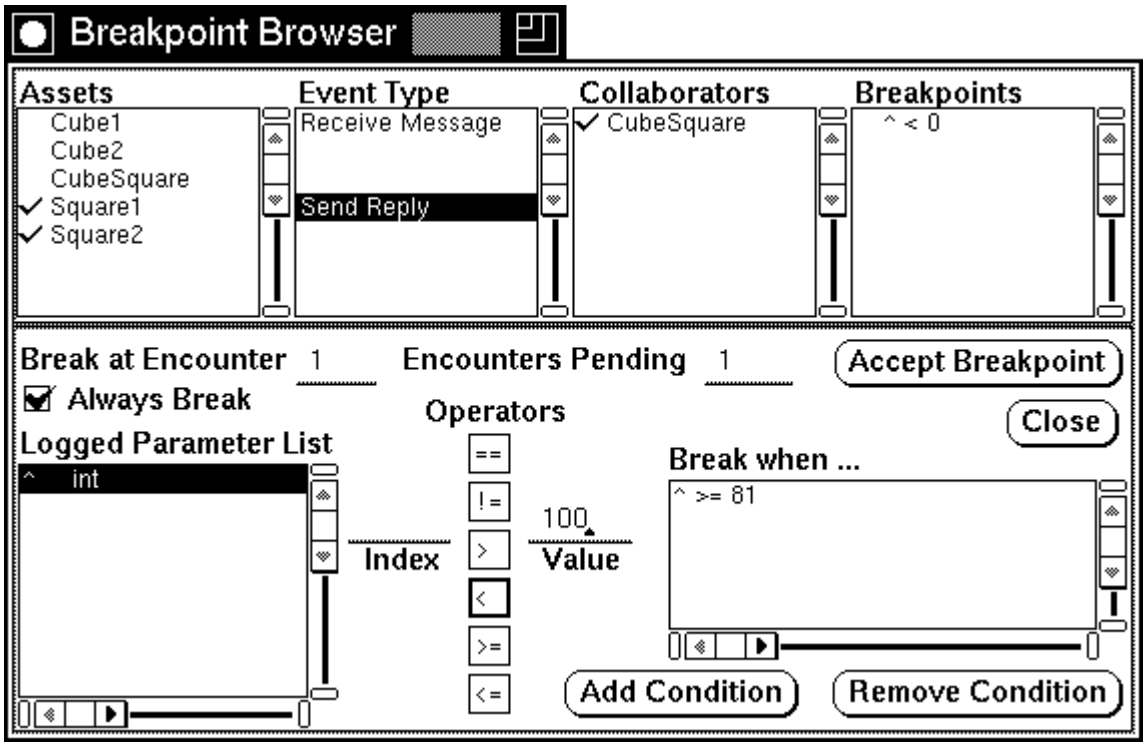


Figure 5.1. Breakpoint Browser.

5.3.1 Navigation Tools

5.3.1.1. Assets List

The first of the list views in the top portion of the Breakpoint Browser is the Assets list. This view always contains an alphabetical list of all the assets in the Replay View using their animation names. An asset's animation name is its base name, with an appended numerical suffix if the asset is a replica. These animation names are unique, regardless of replication nesting, and correspond to the names that appear in the Replay View. Multiple selections can be made from the list by clicking the <select> button when the cursor is over a list item. Selected assets are preceded by a check mark (✓). Only assets with a common base name can be multiply selected. A warning appears if an incompatible multiple selection is attempted. Figure 5.1 shows a Breakpoint Browser with the assets `Square1` and `Square2` selected in the Assets list.

Pressing and holding the <operate> button while the cursor is in the Assets list view produces a pop-up menu for breakpoint deletion and asset selection. These menu options are context-sensitive, varying with the existence of breakpoints and the current selection of assets. The possible menu options are:

Remove all breakpoints

Remove all selected breakpoints
Deselect all
Select all replicas

Remove all breakpoints will remove all the set breakpoints for all the assets in the system. If there are no breakpoints in the system, this menu option will not appear.

Remove all selected breakpoints will remove all the set breakpoints for all the currently selected assets. If no assets are currently selected or if the selected assets have no breakpoints, this menu option will not appear.

Deselect all deselects all the currently selected assets. Alternatively, clicking on a selected asset with the <select> button individually deselects it. The Deselect all option does not appear if no asset is currently selected.

Select all replicas will select all the replicas of a currently selected asset. This option will appear only when an asset is currently selected and replicas of the selected asset are currently unselected.

5.3.1.2. Event Type List

The Event Type list is immediately to the right of the Assets list. This pane is empty if no asset is selected in the Assets list. If an asset is selected, the Event Type list will display the types of message-passing events possible for the selected asset. In other words, an event type that yields no potential collaborators would not be displayed. For example, a Service asset can only have a Receive Message or Send Reply event type; the first asset in a program can only have a Send Message or Receive Reply event type. This event type determination is solely based upon the program's *Enterprise* design as reflected in the asset diagram, not upon the program code or the events actually logged in the event file.

The Event Type list is a list view that allows a single selection. If an unselected item is clicked on, it becomes selected (displayed in reverse video), and any previously selected item becomes deselected. Clicking on a selected item deselects it. The Breakpoint Browser in Figure 5.1 has Send Reply selected in the Event Type list.

The context-sensitive menu for this pane contains only two possible options:

Remove all breakpoints
Remove all selected breakpoints

The behavior of these operations is analogous to that of the Assets list menu options. Removal affects all breakpoints for the selected assets and the pertinent event type(s), irrespective of collaborators.

5.3.1.3. Collaborators List

The Collaborators list, situated immediately to the right of the Event Type list, is empty until an asset and event type have been selected. When an event type has been selected in the Event Type list, all the potential collaborating assets are displayed in the Collaborators list. The rules for collaboration are as follows:

1. For a Line, each asset, except the last, can send a message to the next asset in the Line. The last asset can send a message out of the Line if the Line is embedded in another Line with more assets remaining.

2. For a Department, the Receptionist can send a message to all the other Department assets. All the assets in a Department, including the Receptionist, can send a message out of the Department if the Department is embedded in a Line with remaining assets.

3. For a Division, the Receptionist can send a message to its Representative or Division component. All the assets in a Division can send messages out of the Division if the Division is embedded in a Line with remaining assets.

4. All Receptionists, Individuals, and Representatives can send messages to all Services.

5. If a message-receiving asset is a replicated asset (represented by a manager), the receivers of the message are taken to be all the replicas, rather than the replicated asset itself.

6. If the receiving asset is a composite asset, the collaborator of a Send Message event is the Receptionist of the composite.

7. If a receiving asset is an Individual, Representative, or Service, it becomes a collaborator of the Send Message event.

8. The collaborators for a Received Reply event are the same as those for a Send Message event.

9. The collaborators for a Send Reply event or a Receive Message event are the inverse of those of a Send Message event. That is, if `AssetB` is a Send Message collaborator for `AssetA`, then `AssetA` is a Send Reply collaborator and a Receive Message collaborator for `AssetB`.

At the time a message is sent to a replicated asset, it is not known which replica will ultimately receive the message. Although the ultimate destination could be determined in a replay situation by searching ahead through the event file, it is intuitive and logical to simply list the base name of the replicated asset, rather than all of the replicas individually. Selecting the replicated asset's name then effectively selects all the replicas as collaborators for the purpose of defining a breakpoint.

Like the Assets list, the Collaborators list is a multi-selection list view. All the selection operations and menu choices are identical to those in the Assets list. Breakpoint removal behavior, though not identical, is analogous. Figure 5.1 shows `CubeSquare` selected in the Collaborators list.

5.3.1.4. Breakpoints List

The Breakpoints list, the rightmost breakpoint navigation tool, is empty until a collaborator , or set of collaborators, is selected. Once the collaborator selection is made, an abbreviated list of breakpoints appears in the Breakpoints list if breakpoints exist for the chosen assets/event-type/collaborators selection. A breakpoint is identified and labeled by the first condition in its set of conditions. A breakpoint with no conditions is labeled as Unconditional. When a breakpoint is selected from this list, its full specification is displayed in the lower portion of the browser. This specification can then be edited as described below in Section 5.3.4.

The selection mechanism and menu options are similar to those in the other multi-selection list views. Any number of breakpoints can be selected concurrently. However, if more than one is selected, the information in the lower portion of the browser reverts to the default settings, not a representation of any of the selected breakpoints. Multiple selection is only useful for quick selective removal of set breakpoints. Figure 5.1 illustrates a Breakpoint list with one unselected breakpoint.

There is a subtle feature regarding the removal of breakpoints involving replicas of assets or collaborators. If, for instance, a breakpoint is set for all replicas of an asset and all replicas of a collaborator, it will appear in the Breakpoints list for any sub-group of those replicas of assets and collaborators. If removal is then initiated through the Breakpoint list, the breakpoint will be removed only with regard to the selected assets and collaborators. It will still exist in the system for any unselected assets and collaborators left in its specification. If a breakpoint contains no assets or no collaborators in its specification, it is no longer in the system.

5.3.2. Definition Tools

The lower portion of the Breakpoint Browser contains the breakpoint definition tools. These allow the user to specify the conditions for breaking and the number of times those conditions must be encountered before the breakpoint is actually triggered.

5.3.2.1. The Encounter Setting

The field labeled Break at Encounter specifies the number of times this breakpoint's conditions must be satisfied before the breakpoint is triggered. The default setting is 1. The Encounters Pending field is read only. It is automatically set equal to the value of the Break at Encounter field when the breakpoint is accepted. During replay, when this breakpoint is encountered, the Encounters Pending field counts down by one. When Encounters Pending reaches zero, the breakpoint is triggered.

Once Encounters Pending reaches zero, the breakpoint is no longer active unless the user checked the box Always Break at the time the breakpoint was created. Checking Always Break automatically resets the Encounters Pending to its initial value after the breakpoint has been triggered and passed. Since the breakpoint is triggered and the replay stopped *before* the triggering event is replayed, the system must allow that breakpoint event to actually get replayed when replay is resumed. Therefore, to prevent deadlock, Encounters Pending is not re-initialized until after the event is executed.

5.3.2.2. Constructing Conditions

As previously indicated a breakpoint need not have any conditions attached to it. Such a breakpoint, labeled as Unconditional in the Breakpoints list, will be triggered on the n^{th} encounter of an event involving the selected assets, event type, and collaborators, where n is the value set for Break at Encounter.

A breakpoint can have any number of breakpoint conditions attached to it. Each condition is described by selecting a parameter from the Logged Parameter List, selecting an operator from the Operators radio buttons, entering an appropriate value in the Value field, and, if necessary, entering an integer index in the Index field.

The parameters logged in the event file for the event type in question are listed in the Logged Parameter List using ① the parameter's name from the argument list of the called asset, ② a description of the parameter's *logged elements*, if the parameter is an array, and ③ a description of the parameter's type. The return value of an asset call, which is unnamed, is identified by the caret symbol (^). The logged elements notation for

arrays describes the indices of the array values which have been logged. The syntax of this notation is specified in the BNF for <ranges> in Appendix A. Following the C convention, the first element of an array is indexed by 0. Negative index values denote elements referenced from the end of the array. Thus, -1 refers to the last element; -2, the next to last. A range can be represented by a pair of indices in parentheses ( ). The logged elements notation allows any number of single indices and ranges to be included within square brackets [ ]. For example,

y[(0 2) 5 (-3 -1)] array of int

denotes that the logged elements (integer values) of the array y are namely the first three elements, the *sixth* element, and the last three elements. The notation

y[(0 -2)] array of char

could be used for logging a character string in C, omitting the terminating null byte, assuming that y is dynamically allocated to be the size of the string it contains.

To reference a value in an array, the user can enter an integer in the Index field. If an index is entered for a scalar, or if the index entered for an array is incompatible with the logged elements notation, attempting to add the condition to the list of conditions will generate an error warning. The invalid condition will not be added to the list. Since arrays can be dynamically allocated and of variable length, there is no way to check if an index is out of bounds when any of the *logged elements* or the index itself is referenced from the end of the array.

The relational operators (==, !=, >, <, >=, and <=) appear as radio buttons. Only one operator can be selected at a time.

To construct a valid condition, the programmer must make a valid entry in the Value field. If the type of the selected parameter is *int*, an integer (no decimal point) must be entered. For a *float*, the decimal point is optional. Scientific notation, as specified for <float> in Appendix A, can also be used. For a *char*, the user can enter a printable character or the decimal ASCII code preceded by a backslash (\). If the logged parameter is a single range of array of char, a string of characters can be entered in the Value field. This string may contain non-printable characters written as a backslash followed by three digits representing decimal ASCII code. A backslash character in a string is represented by two backslashes.

This mechanism of constructing a condition through list view selection, radio button selection, and input field validation makes it simple for the user to quickly create legitimate breakpoints. The possibility of introducing errors is minimized

5.3.2.2. Adding and Removing Conditions

Once a condition has been constructed, it can be added to the breakpoint's conditions by clicking on the Add Condition button in the Breakpoint Browser. If the condition is completely and validly constructed, it will be added to the list view labeled Break when . . . . Otherwise, an appropriate warning message appears. Any number of conditions can be added in this manner. All the conditions in the list view are logically ANDed. If an OR relationship is required, separate breakpoints can be set up to achieve this relationship. This requirement is not restrictive: logically, any combination of ANDs and ORs can be reduced to disjunctive normal form, a disjunction of conjunctions of elementary conditions.

If a condition added to the Break when . . . list is not desired, it can be removed by selecting it and then clicking on the Remove Condition button in the browser.

5.3.3. Accepting Breakpoints

After all the desired conditions have been added and the encounter setting made, the breakpoint can be accepted by clicking on the Accept Breakpoint button. When a breakpoint is accepted, three things occur: ① the first condition of the breakpoint (or Unconditional) will appear unselected in the Breakpoints list; ② the breakpoint settings in the Browser will revert to the defaults; and ③ an icon of a stop sign will appear beside the appropriate asset icons in the graphical replay view.

A breakpoint does not become an active part of the system until the Accept Breakpoint button is clicked.

5.3.4. Editing Breakpoints

When a single set breakpoint is selected in the Breakpoints list, the particulars of the breakpoint are displayed for editing. The encounter setting can be changed and conditions may be added or removed. When an edited breakpoint is accepted, the operation is equivalent to removing the previously set breakpoint and accepting the currently defined breakpoint.

As pointed out above in Section 5.3.1.4, when multiple breakpoints are selected from the Breakpoints list, the breakpoint definition portion of the browser reverts to default values. Breakpoint acceptance is not allowed in this circumstance. All breakpoints in the Breakpoints list should first be deselected before defining and accepting a new breakpoint.

A breakpoint can be deactivated, rather than removed, by setting the Break at Encounter field to 0, ensuring that the Always Break box is not checked, and then accepting the breakpoint. This results in retaining the breakpoint and all its conditions in the system, but the breakpoint will never be triggered. Such a deactivated breakpoint can easily be reactivated by again editing the Break at Encounter field and accepting the modified breakpoint.

5.4. Graphical Display of Breakpoints

The management of breakpoints is simplified by the coordination among the display in the Replay View, the contents of the Breakpoint Browser, and the breakpoint state of the underlying model.

5.4.1. Description of Symbols

Any time a breakpoint is set, its existence in the system is portrayed as a red (black in monochrome displays) stop sign icon at a corner of the asset icon with which it is associated. The relative position of the breakpoint icon and a directional arrow in its center identify the type of event to which the breakpoint is responsive:

Send Message (lower left corner of asset icon)

Receive Message (upper left corner of asset icon)

Send Reply (upper right corner of asset icon)

Receive Reply (lower right corner of asset icon)

The presence of a stop sign icon indicates that one or more breakpoints of that particular event type are associated with the particular asset.

5.4.2. Highlighting during Breakpoint Setup

When an asset is selected in a Breakpoint Browser, its icon is highlighted with a wide red (black in monochrome) border in the Replay View. The zones for the breakpoint icons, which correspond to valid event types for a highlighted asset, are also outlined with a fine black line. Figure 5.2 shows `Square1` and `Square2` selected as assets.

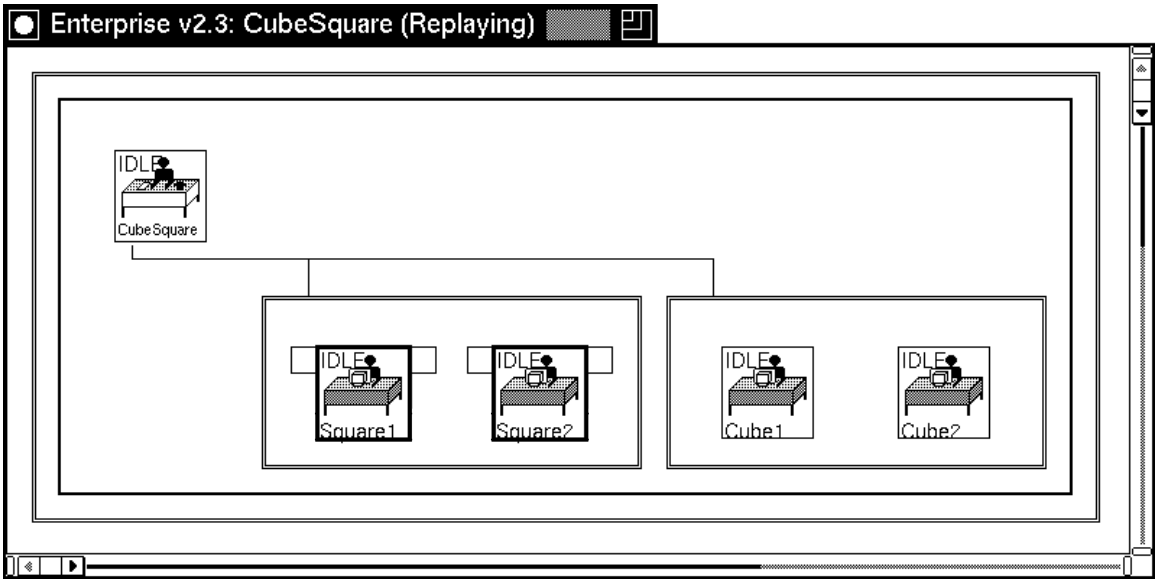


Figure 5.2. Replay View of `CubeSquare` with assets selected.

When a event type is selected, the breakpoint icon zone associated with that event type is highlighted with a wide black line. The unselected breakpoint icon zones lose their outlines. Figure 5.3 shows the replicas of `Square` with the Send Reply event type selected.

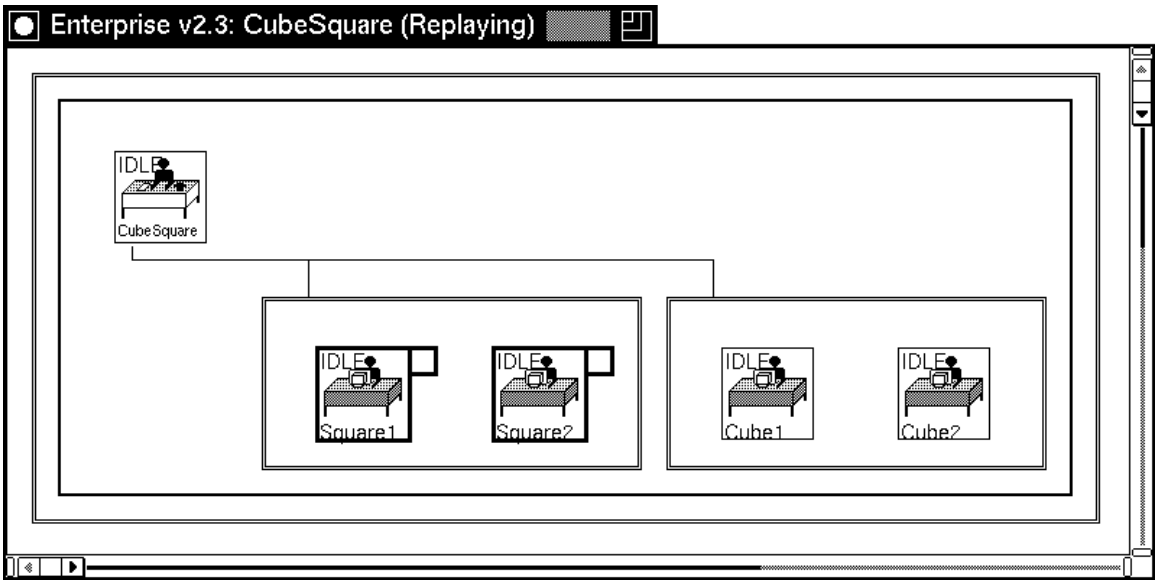


Figure 5.3. Replay View of `CubeSquare` with assets and event type selected.

When a collaborator is selected, a wide red (black in monochrome) line appears between the selected asset's breakpoint icon zone and the collaborator, suggesting the message path that would be followed from asset to collaborator, or vice versa. Multiple selections of assets or collaborators are displayed accordingly. Figure 5.4 illustrates this next step with `CubeSquare` selected as the collaborator for a breakpoint setup.

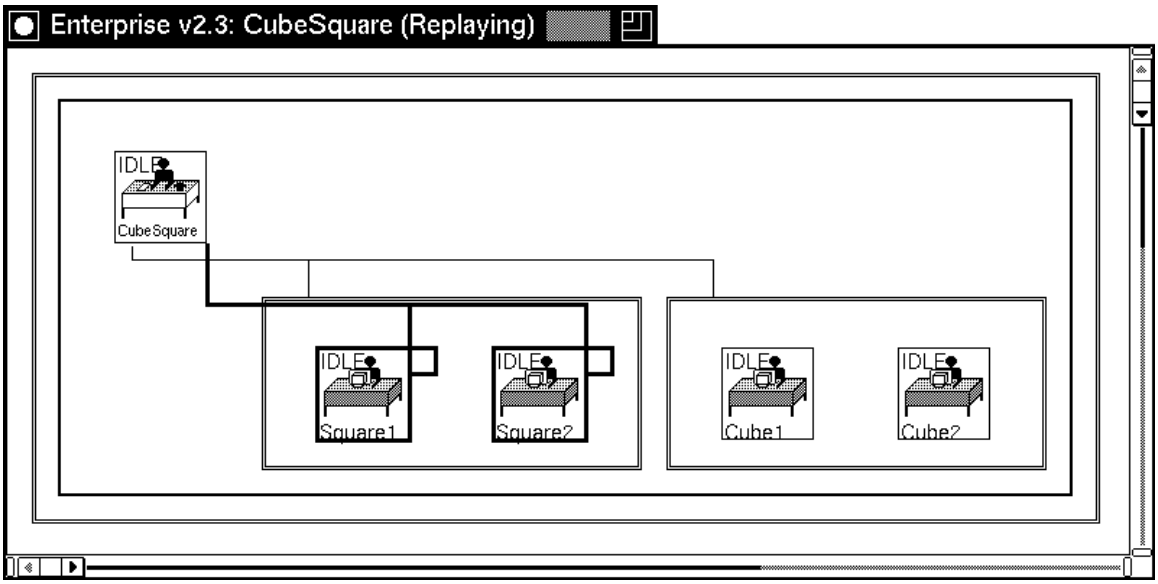


Figure 5.4. Replay View of `CubeSquare` with assets, event type, and collaborator selected.

Figure 5.5 shows a program displaying the highlighting that corresponds with the selections in the Breakpoint Browser in Figure 5.1. To illustrate the different breakpoint icons, breakpoints have been set for various assets using all four types of message-passing events. `CubeSquare` has breakpoints set for Send Message and Receive Reply events. The `Square` replicas have breakpoints set on Send Reply events, and the `Cube` replicas on Receive Message events.

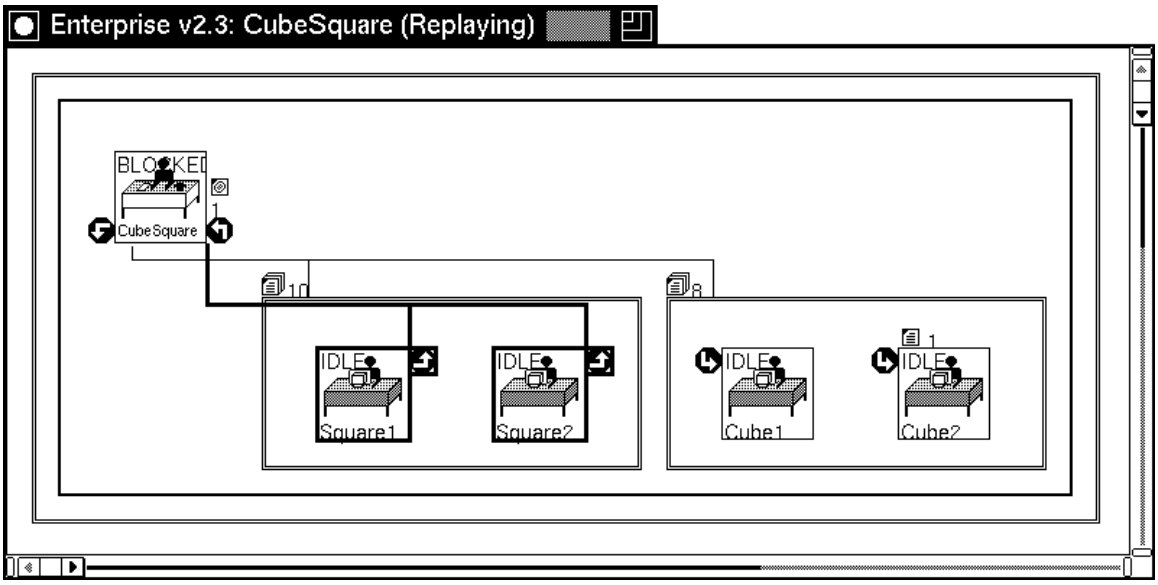


Figure 5.5. Replay View with breakpoints set.

5.4.3. Graphical Control of the Breakpoint Browser

Browser selections can be controlled by selecting icons in the Replay View, rather than items in the browser's navigation tools. An asset can be selected by clicking the <select> mouse button when the cursor is over the asset icon. Multiple asset selections are made by holding down the SHIFT key while making the selection.

The event type can then be selected by clicking in the appropriate outlined breakpoint icon zone. Finally, any number of collaborators can be selected by holding down the CTRL key while making the selection.

Any breakpoint that is set in the system can be selected by clicking on its associated breakpoint icon. If more than one breakpoint is associated with an icon, the user can repeatedly click on the icon to cycle through all the set breakpoints. As each breakpoint is selected, all its associated assets and message paths to collaborators are appropriately highlighted and selected in the browser's list views.

The system allows any number of Breakpoint Browsers to be opened simultaneously. But only one browser is considered active at any moment. The active browser is the last browser to be opened or the last to have had any selection made in its navigation list views or any of its action buttons pressed. Only the active browser has its state reflected in the display of the Replay View. Any time a selection is made in the replay view, it will be reflected in only the active browser. If an active browser is closed, the oldest remaining browser becomes the new active one. If no browser is active when a selection is made in the Replay View, a new browser will automatically be created.

5.5. Breakpoint Triggering

When a program is being replayed, each message-passing event in the event file is examined against the set breakpoints before the *Enterprise* executive is instructed to execute the event. Replay is initiated by selecting Start or Step from the Replay View <operate> menu. Step will process only one event from the event file. If this event is a Start Event, Die Event, or message-passing event, it will result in an instruction to the executive to proceed until another Die Event or message-passing event is imminent. If however an active breakpoint has all its conditions satisfied and its Encounters Pending is 1, the breakpoint will trigger and the event will not execute. Upon triggering, the responsible breakpoint will be highlighted in the Replay View, the active Breakpoint Browser will pop to the foreground (or be created if necessary), and the triggering breakpoint's specification will be displayed in it. The identifying characteristics of the message causing the breakpoint to trigger (receiver, sender, and tag number) are displayed in the System Transcript.

If the event type is Receive Message or Receive Reply, the specific message which triggered the breakpoint is still in the asset's input or reply queue. From the information in the System Transcript, the user can open a Message Queue List on the appropriate message queue (using the <operate> button on the message queue icon) and then select the relevant message from the list view in the top pane of the Message Queue List. The message's logged parameters will then all be displayed in the text view located in the lower pane.

If the event type is Send Message or Send Reply, the user can single step through the triggering event, allowing the triggering message to enter its appropriate message queue. Once there, it can be examined in detail.

In the event that unlogged message parameters or local process state information is desired, the programmer may open a sequential debugger on an asset's process by selecting Sequential Debugger from the menu associated with the target asset.

5.6. Implementation Issues

Breakpoints are implemented as Smalltalk objects that are instances of the class Breakpoint, defined as a subclass of Model. A breakpoint has the following instance variables: *assets*, *type*, *collaborators*, *conditions*, *bpCounterInitial*, *bpCounterCurrent*, and *alwaysBreak*.

The *assets* variable is a set of (pointers to)[1] all the assets that the breakpoint belongs to. Each asset, conversely, has a *breakpoints* instance variable which is a collection of all the breakpoints belonging to that asset. Similarly, the *collaborators* variable is a set of all the collaborators associated with the breakpoint. The *type* is an integer identifying the event type of the breakpoint.

The *bpCounterInitial* is the setting that the user enters in the Break at Encounter field. The *bpCounterCurrent* is the value that appears in the Encounters Pending field. Initially, *bpCounterCurrent* is set equal to *bpCounterInitial*. The *alwaysBreak* is a Boolean to indicate if the user has checked the Always Break check box.

The *conditions* is a collection, possibly empty, of instances of BreakpointCondition, a subclass of Model. BreakpointCondition has instance variables: *parameter*, *index*, *operator*, and *bpValue*. These BreakpointCondition objects represent, in an obvious manner, each of the conditions that the user has added to the breakpoint's definition with the Breakpoint Browser.

Before any message-passing event is sent to the *Enterprise* executive for replay, it is first examined to see if it satisfies any of the relevant set breakpoints. If a breakpoint is indeed satisfied, its *bpCounterCurrent* is decremented by 1. If the counter's value reaches zero, the breakpoint is triggered, preventing the replay instruction from being sent to the executive. Special considerations are made for triggered breakpoints whose *alwaysBreak* is true, so that the breakpoint is reset (by again making *bpCounterCurrent* equal to *bpCounterInitial*) after the triggering event has eventually been replayed.

---

[1]Objects in Smalltalk are referenced by pointers. Hereafter, this will not be expressed explicitly.

Chapter 6

Conclusion

6.1. Individual vs. Group Contributions

Enterprise is a large software development effort. The work of this researcher was not conducted in a vacuum. Weekly project meetings with the full *Enterprise* team, weekly thesis meetings with supervisors, periodic design meetings with smaller working groups, and nearly daily interaction with other *Enterprise* researchers all point to the fact that the work described here is the product of many minds and many hands. In particular, design issues were reviewed by various individuals at project and thesis meetings. Many of the ideas and suggestions were eventually incorporated in the present design.

The research for this thesis included the following direct contributions to the *Enterprise* PPS:

1. Design of the replay algorithm.
2. Modification of the *Enterprise* executive to implement the replay algorithm.
3. Redesign of the event-logging format.
4. Specification of a editable parameter logging mechanism for the executive with compiler support.
5. Complete design and implementation of the interface for parameter logging.
6. Complete design of the debugging system.
7. Redesign and re-implementation of the basic animation algorithm.
8. Implementation of the debugging system in the interface based on the modified animation system.
9. Integration of the new components into a unified system.

This researcher also had the responsibility of general maintenance of the interface during much of the year.

6.2. Future Work

Many of the features mentioned in Appendix B are currently being implemented. Since required modifications to the pre-compiler, executive, and interface have only recently been implemented by various members of the *Enterprise* team, it has not been possible to fully test the breakpointing mechanism on a wide range of programs. Preliminary testing has been successfully done using a suite of "real" programs [Par93] (more than 1,000 lines of code), unlike the "toy" programs **`CubeSquare`** and **`ManShootsDog`** used for illustrative purposes here. Comprehensive testing is an essential part of the work in progress.

The following enhancements and extensions to the *Enterprise* debugging facilities are under development or have been proposed:

- The current version of the *Enterprise* executive does not allow for logging fields in structures. A more general means of handling structures, including self-referential structures, is a focus of the new pre-compiler and executive designs. The pre-compiler, executive, Parameter Logging Editor and Breakpoint Browser will require coordinated modifications when structure logging is implemented.

- The breakpointing facility could easily be extended to the Animation View to control animation during a simulation of a program run. This would allow the user to easily examine parameter and return values when certain breakpoint conditions were satisfied without necessitating a re-execution of the program, which might be computationally intensive, coarse-grained, and of long duration. This already works for breakpoints on non-replicated assets.

- As the *Enterprise* performance monitoring and analysis tools become more refined, breakpointing might be extended to encompass performance conditions. The user could then break, for instance, when a bottleneck had occurred at a particular asset, when an asset's idle time had exceeded some threshold, or when a message size had exceeded a specified limit.

- Another useful extension would be to define breakpoints in terms of a combination of events within a particular asset call or between asset calls. This would be of advantage, say, in a parallel recursion tree where we might be interested in breaking when an input parameter satisfies some condition on a Receive Message event *and* a return parameter satisfies some other condition on a Send Reply during the same asset call. This extension would move the single-event definition language developed in this thesis to a higher level multi-event definition language, where various confluences of events could be described. Such event definition languages have been implemented in other systems [BW83] [BH83].

- As the system is now designed, a sequential debugger attached to a running asset will display source code as it appears *after* pre-compiler modifications. To avoid the confusion of the user viewing *Enterprise* system code intermingled with user-defined code, it would be possible for the pre-compiler to flag the *Enterprise* code and to modify the sequential debugger to display only the unflagged user source code. This would eliminate unnecessary clutter from the user's view of his code's execution. A switch could be used to include system code in the debugger display if desired.

- It is also possible to extend the debugging capabilities to initial runs executed in the Design View before any event file is captured. This could be done in a manner quite similar to the breakpoint implementation in the Replay View. Hooks could be placed in the *Enterprise* executive just before message sends and after message receptions, with descriptions of spied on events sent to the interface for breakpoint analysis. Such an approach would not require symbol table information in the executive.

- The development of a checkpointing and rollback facility similar to that used in DPD would facilitate debugging computationally intensive runs. Re-execution would not always have to begin from the start of the program, as is now the case. It is not clear how much of a drag this would be on the system due to the time and storage requirements of distributed snapshots.

6.3. Scalability

The scalability of this debugging facility is an issue deserving attention. The granularity of *Enterprise* asset calls should be coarse. The overhead of message passing will outweigh the gains of parallel execution if the granularity is too fine. This relates to the scalability in terms of the number of events in an event log. The processing time for the breakpoint check and replay of a single message-passing event is independent of the total number of events. Therefore, the relationship between the number of events and this processing time is linear. An *Enterprise* program with an appropriate granularity will not incur excessive overhead during replay. Inappropriately fine granularity results in poor performance (low speedups or even slow-downs), excessively large event logs, and extended replay times.

Scalability can also be viewed in terms of the number of *Enterprise* processes. The number of processes in an *Enterprise* program is a function of the number of assets and the amount of replication. A typical *Enterprise* program has just a few assets potentially replicated many times. Again the processing time for any single event will not increase significantly as the number of processes increases. But with large replication factors, the display will become exceedingly large. The ability to collapse composite and replicated

assets during animation and replay is planned but not yet implemented. As for setting up breakpoints, that can easily be accomplished by selecting all of an asset's replicas.

Yet another dimension of scalability is total execution time. Without a checkpointing and rollback scheme, the *Enterprise* debugger is of limited value for programs with long execution times. In such a situation, it would be preferable to log relevant message data and simulate the execution in the Animation View. This will become more practical when the breakpointing control is extended to the Animation View in a future release.

6.4. Conclusion

This thesis has presented a practical, easy-to-use event-based debugger for *Enterprise* programs. It confirms the hypothesis that deterministic execution replay and event-level debugging are compatible with the constraints of the *Enterprise* model. The approach has combined and coordinated graphical displays with textual information to provide the user with a high-level intuitive perception of the program's progress together with detailed information about parameter values in specific asset calls. The menu, list, radio-button, and check-box techniques used in specifying parameter logging and breakpointing make it virtually impossible for the user to make illegal or inappropriate responses. In the few places where textual input is required, data is type-checked to determine its suitability for the entry.

ParcPlace VisualWorks has proven to be a highly effective medium for the graphical interface development. Building the visual display of complex dialog windows, like the Breakpoint Browser, is fast and uncomplicated in VisualWorks. The coordination of views based on the same model is also greatly facilitated by the mechanisms provided in Smalltalk. For the incremental development of evolving prototypes like *Enterprise*, the advantages of Smalltalk, with its object-oriented paradigm and fully integrated programming environment, are well known. The experience of developing the *Enterprise* interface, with its frequently modified specifications, has amply demonstrated these advantages. Though the system has become quite large and complex (about 13,000 lines of Smalltalk code in the interface alone), changes have been easy to make without extensive re-coding, and programming errors have usually been straight-forward to debug. Such flexibility and ease of development is unmatched in other programming systems. Processing speed in Smalltalk, a commonly voiced concern about this interpreted, dynamically-typed language, has generally been found to be acceptable. The primary disadvantage is that the interface cannot be shipped as a stand-alone executable and used on a machine that is not already licensed to run VisualWorks. But the cost of creating an equivalent X-Windows version is prohibitive

As yet, the *Enterprise* user community is small. The system has received greatest attention in Europe, where it is being evaluated by Siemens as a model for developing

template-based parallel computing. More programs need to be developed in *Enterprise* by a wider group of users before the true benefits of these debugging facilities can be adequately assessed. It is hoped that *Enterprise*, with these new tools in place, can help span the gap between the sequential and parallel computing worlds for the growing number of programmers working with an enormous amount of under-utilized computing power at their disposal. The arduous chore of distributed debugging will be hopefully lessened. For anyone working with computers, *gain without pain* is indeed a noble pursuit.

Bibliography

[AR94] Joshua S. Auerbach and James R. Russell. "The Concert Signature Representation: IDL as Intermediate Language," *Proceedings of the 1994 ACM SIGPLAN Workshop on Interface Definition Languages* (1994).

[BFV86] Fabrizio Baiardi, Nicoletta de Francisco, and Gigliola Vaglini. "Development of a Debugger for a Concurrent Language," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 4, pp. 547-553 (April 1986).

[BW83] Peter C. Bates and Jack C. Wileden. "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach," *Journal of Systems and Software*, Vol. 3, No. 4, pp. 255-264 (1983).

[Bat89] Peter Bates. "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior," *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, published in *ACM SIGPLAN Notices*, Vol. 24, No. 1, pp. 11-22 (January 1989).

[BDG91] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Vaidy Sunderam. "A Users' Guide to PVM Parallel Virtual Machine," Oak Ridge National Laboratory, Technical Report ORNL/TM-11826 (July 1991).

[BDG93] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Keith Moore. "HeNCE: A Heterogeneous Network Computing Environment," Carnegie Mellon University, Computing Science Department, Technical Report CS-93-205 (August 1993).

[BDG94] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Vaidy Sunderam. "HeNCE: A Users' Guide Version 2.0," Oak Ridge National Laboratories, `hence@epm.ornl.gov` (February 1994)

[BBG94] François Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. "Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools," to appear in *The Proceedings of the Second Annual Object-Oriented Numerics Conference (OON-SKI '94)* (1994). Available from World Wide Web at `http://www.cica.indiana.edu/sage/docs.html`.

[BH83] Bernd Bruegge and Peter Hibbard. "Generalized Path Expressions: A High Level Debugging Mechanism," *Journal of Systems and Software*, Vol. 3 , No. 4, pp. 265-276 (1983).

[CL85] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, Vol. 3, No. 1, pp. 63-75 (February 1985).

[CBM87] W. H. Cheung, James P. Black, and Eric G. Manning. "A Study of Distributed Debugging," Research Report CS-87-53, University of Waterloo, Faculty of Mathematics (October 1987).

[DF93] Suresh K. Damodaren-Kamal and Joan M. Francione. "Nondeterminacy: Testing and Debugging Message Passing Parallel Programs," *ACM SIGPLAN Notices*, Vol. 28, No. 12, pp. 118-128 (December 1993)

[Fid89] Colin J. Fidge. "Partial Orders for Parallel Debugging," *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, published in *ACM SIGPLAN Notices*, Vol. 24, No. 1, pp. 183-194 (January 1989).

[FFG94] M. Fruscione, P. Flocchini, E. Giudici, S. Punzi, and P. Stofella. "Parallel Computational Frames: An Approach to Parallel Application Development Based on Message Passing Systems," *Preprints of the International Federation for Information Processing (IFIP) Working Conference on Programming Environments for Massively Parallel Distributed Systems*, Monte Verità, Switzerland, pp. 12-1 to 12-10 (April 1994).

[GGK84] Hector Garcia-Molina, Frank Germano, Jr., and Walter H. Kohler. "Debugging a Distributed Computing System," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 2, pp. 210-219 (March 1984).

[JZ88] David B. Johnson and Willy Zwaenepoel. "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing," *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, pp. 171-181 (May 1988).

[LM87] Thomas J. LeBlanc and John M. Mellor-Crummey. "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers*, Vol. C-36, No. 4, pp. 471-482 (April 1987).

[Lob93] Greg Lobe. *The Enterprise User Interface and Program Animation Component*, M. Sc. thesis, Department of Computing Science, University of Alberta (1993).

[MBS88] T. A. Marsland, T. Breitkreutz, and S. Sutphen. "NMP—A Network Multi-Processor," Technical Report TR 88-22 (December 1988).

[MH89] Charles E. McDowell and David P. Helmbold. "Debugging Concurrent Programs," *ACM Computing Surveys*, Vol. 21, No. 4, pp. 593-622 (December 1989).

[PU91] Cherri M. Pancake and Sue Utter. "A Bibliography of Parallel Debuggers, 1990 Edition," *ACM SIGPLAN Notices*, Vol. 26, No. 1, pp. 21-37 (January 1991).

[PN93] Cherri M. Pancake and Robert H. B. Netzer. "A Bibliography of Parallel Debuggers, 1993 Edition," *ACM SIGPLAN Notices*, Vol. 28, No. 12, pp. 169-186 (December 1993).

[Par93] Ian Parsons. *An Appraisal of the Enterprise Model*, M. Sc. thesis, Department of Computing Science, University of Alberta (1993)

[PL83] Michael L. Powell and Mark A. Linton. "A Database Model for Debugging," *Journal of Systems and Software*, Vol. 3 , No. 4, pp. 295-300 (1983).

[SSL93] Jonathan Schaeffer, Duane Szafron, Greg Lobe, Ian Parsons. "The Enterprise Model for Developing Distributed Applications," *IEEE Parallel and Distributed Technology*, Vol. 1, No. 3 (August 1993).

[SS89] R. S. Side and G. C. Shoja. "DPD: A Distributed Program Debugger for the REM Environment," *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, pp. 265-268 (June 1989).

[SS94a] R.S. Side and G.C. Shoja. "A Debugger for Distributed Programs," *Software Practice and Experience*, Vol. 24, No. 5, pp. 507-525 (May 1994).

[SS94b] Duane Szafron and Jonathan Schaeffer. "Experimentally Assessing the Usability of Parallel Programming Systems," *Preprints of the International Federation for Information Processing (IFIP) Working Conference on Programming Environments for Massively Parallel Distributed Systems*, Monte Verità, Switzerland, pp. 19-1 to 19-7 (April 1994).

[SSI94] D. Szafron, J. Schaeffer, P. Iglinski, I. Parsons, R. Kornelsen, and C. Morrow. "Enterprise: Current Status and Future Directions," submitted to *CASCON 94* (1994).

[Tay92] David Taylor. "A Prototype Debugger for Hermes," *Proceedings of the 1992 CAS Conference (CASCON 92)*, Vol. 1, pp. 29-42 (November 9-12, 1992).

Appendix A

New *Enterprise* File Formats

This appendix uses extended BNF (Backus-Naur Form) notation to describe the file formats referenced in this thesis.

A.1. Event Log File

When the Event logging box is checked in the Run Dialog Box before a program is run, an event file is generated in **./*ProgName*/`Graph`/*ProgName*.`ev`**.

A.1.1. Notation

< > is used to enclose the name of a non-terminal syntactic category.
::= means "is defined as."
↵ designates a newline.
␣ designates a single space.
***** (the *Kleene star*) designates repetition of the preceding element zero or more times.
+ (the *Kleene cross*) designates repetition of the preceding element one or more times.
| indicates an OR relationship between the preceding and following expressions.
{ } is used to group expressions.
[ ] is used to enclose an optional expression.
' ' indicates that the enclosed symbol appears literally as a terminal symbol.

A.1.2. Syntax

<event log> ::= <header> <event>*

<header> ::= '{' ↵ <asset info>+ '}' ↵

<asset info> ::= <asset name> **:** ↵
[<input params> **]**
[<output params>**]** **.** ↵

<input params> ::= '>' ↵ <logged param>+

<output params> ::= '<' ↵ <logged param>+

<logged param> ::= {<param name> | ^} ^ <type> [<ranges>] ↵

<asset name> is any legal C function name used as an asset name in the program.

<param name> is any legal C variable name used in the parameter list of the asset.

<type> ::= % <type code> ^

<type code> ::= [P] {d | f | c}

<ranges> ::= '[' <range>+ ']'

<range> ::= {<int> | (<int> ^ <int>)} ^

<int> ::= {[-] <pos int>} | 0

<pos int> ::= <non-zero digit> <digit>*

<non-zero digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<digit> ::= <non-zero digit> | 0

<event> ::= <event description> ↵ <param record>*

<param record> ::= { <param value> | <size> ^ <param value>* } ↵

<param value> ::= { <int> | <float> | <char> | OOR | NaN | Infinity } ^

<size> ::= <pos int> | NIL

<float> ::= <int> [. <pos int>] [{ E | e } <int>]

<char> ::= 0 | 1 | . . . | 255

A.1.3. Semantics

<event log> The <event log> is the entire event file. It contains a header describing the logged parameters and all the events that were logged during a program run.

<header> The <header> contains a description of the logged parameters for each of the program's assets. The contents of the <header> are enclosed in curly brackets ({ }), each bracket followed by a newline.

<asset info> The <asset info> contains the parameter logging specification for a particular asset. It begins with the asset's name, followed by a colon (:) and newline, followed by the input parameters (if any are logged), followed by the output parameters (if any are logged), followed by a period (.) and a newline.

<input params> The <input params> begin with a *greater than* symbol (>) and a newline, followed by a series of one or more logged parameter descriptions.

<output params> The <output params> begin with a *less than* symbol (<) and a newline, followed by a series of one or more logged parameter descriptions.

<logged param> The <logged param> is a description of a logged parameter. It begins with the parameter name or the caret symbol (^), followed by a space, followed by the type, followed by the ranges to be logged (if the type is a pointer), followed by a newline.

<type> The <type> is a percent symbol (%), followed by a <type code>, followed by a space.

<type code> The <type code> specifies the type of the logged parameter. If the parameter is a pointer, the <type code> begins with a 'P'. The letter 'd' refers to an integer type; the letter 'f' refers to a floating point type; the letter 'c' refers to a character type. Thus the <type code> 'Pc' indicates that the parameter is a pointer to characters, potentially a string.

<ranges> <ranges> is a list of one or more <range> enclosed in square brackets ([ ]). The <ranges> specify which elements in an array have been logged.

<range> A <range> is either a single integer or a pair of integers separated by a space and enclosed in parentheses (( )). Each <range> ends with a space. Following the C convention, 0 refers to the first element in an array. Negative integers denote elements at the end of an array. Thus, -1 refers to the last element of an array. The <range> (0 -1) denotes that the entire array is logged.

<int> <int> is any integer, as normally defined.

<pos int> <pos int> is any non-zero positive integer.

<non-zero digit> <non-zero digit> is any digit from 1 to 9.

<digit> <digit> is any <non-zero digit> or 0.

<event> An <event> represents any message passing event or state change event recorded during a program run. The <event> begins with an <event description>[1], followed by a newline, followed by zero or more <param record> of the logged parameters. The only events containing logged parameters are #sentMsg and #sentReply. The number of <param record> must agree with the number of parameters listed in the header for this asset (<input params> for a #sentMsg event and <output params> for a #sentReply).

<param record> A <param record> is the record of a logged parameter. If the parameter is a scalar, this is a single <param value>. For vectors, the first item recorded is <size>, followed by a space, followed by zero or more <param value>. The number of <param value> for vectors must agree with the number of array elements indicated in the header for the parameter. If one of the indicated ranges is dynamic (i.e. (0 -1)), the number is calculated from the <size>. A <param record> ends with a newline.

---

[1]A detailed BNF for <event description> can be found in Appendix B in [Lob93]. These details are not relevant in this context.

<param value> A <param value> can be an integer, a float, a character, or one of the following special values: OOR, NaN, Infinity.
OOR (Out Of Range) indicates that the index requested for logging is not in the range of the actual parameter. NaN (Not a Number) and Infinity have the standard meaning in UNIX.
A <param value> ends with a space.

<size> The <size> is either a positive integer denoting the number of elements in an array or the special symbol NIL. NIL denotes that the parameter indicated for logging was not included in message passed. For example, if values in the vector `y` were requested to be logged as OUT parameters, but `y` was designated as an IN parameter on that particular call, then NIL would appear in that event for the <size>. If the <size> is NIL, no <param value> will be listed.

<float> A float may appear in decimal or scientific notation.

<char> A character is represented by its ASCII code.

A.1.4. Examples

Example of event file header:

```
{
CubeSquare:
.
Square:
>
x^%d^
y^%f^[(1^3)^5^(-3^-1)^]
<
y^%f^[10^]
^^%d^
.
Cube:
<
^^%d^
.
}
```

This header contains the logging specification for a program with three assets: **CubeSquare**, **Square**, and **Cube**. **CubeSquare**, being the first asset in the program, has no parameters logged. **Square** has an integer `x` logged on input, as well as seven elements from the array of floats, `y`, corresponding to `y[1]`, `y[2]`, `y[3]`, `y[5]`, and the last three elements in `y`. Upon return, `y[10]` and the function's return value, an integer designated by the symbol `^`. **Cube** has no parameters logged when the asset is called, but the integer return value is logged when the asset returns.[2]

Example of some event file events:

```
#sentMsg CubeSquare.1 Square.1.1 . . .
5^
20^1.0^1.0^1.0^1.0^1.0^1.0^1.0^
#sentReply Square.1.1 CubeSquare.1 . . .
NIL^
25^
#sentReply Cube.1.1 CubeSquare.1 . . .
125^
```

---

[2]This event file does not correspond exactly to the **CubeSquare** program shown in Chapter 2. The array `y` is added here to illustrate the features of array logging.

The first event, a #sentMsg event from **CubeSquare** to **Square1**[3], has two logged parameters corresponding to the <input params> for **Square**. **Square1** was called with x equal to 5 and the array y, consisting of 20 elements, had each of its seven logged elements equal to 1.0.

The next event, a #sentReply from **Square1** to **CubeSquare**, indicates that y was not logged when the asset returned (NIL) since y was an IN parameter on this particular call to **Square**. The return value was 25.

In the last event, a #sentReply from **Cube1** to **CubeSquare**, the return value was 125.

[3]The numbering suffix for assets that appears in the event file differs from what the user sees in the interface. The rules for generating these suffixes can be found in [Lob93].

A.2. Default Format and Format Files

When an asset is compiled a default format file is generated in ***./ProgName*/`Graph`/*AssetName*`.fmt.dflt`**. This file can be modified using the Parameter Logging Editor. The result appears in ***./ProgName*/`Graph`/*AssetName*`.fmt`**.

A.2.1. Notation

The notations and BNF specifications in A.1. also apply here.

A.2.2. Syntax

<call file> ::= <param info>*

<param info> ::= {<param name> | ^} ^ <param type> ^ <log spec> ↵

<param type> ::= % <param type code> <bytes>

<param type code> ::= [P] { d | f | c | v | R }

<bytes> ::= <pos int> | 0

<log spec> ::= <log code> [<ranges>]

<log code> ::= X | I | O

A.2.3. Semantics

<call file> — The <call file>, one for each asset in the program, contains a description and logging specification, the <param info>, for every parameter in the asset's parameter list and for the return value.

<param info> — The <param info> contains the parameter name or the caret symbol (^) for a return value, followed by a space, followed by the parameter type, followed by a space, followed by the logging specification, followed by a newline.

<param type> The <param type> consists of a percent symbol (%), followed by a code for the parameter type, followed by the number of bytes in each element of the parameter.

<param type code> The <param type code> is much like the <type code> above, but with two additional codes, 'v' and 'R'. The letter codes have the following meanings: 'P'—pointer; 'd'—short, int, or long; 'f'—float or double; 'c'—char; 'v'—void; 'R'—structure (record) or any type not of the above.

<bytes> <bytes> is a positive integer or zero denoting the number of bytes in one of the parameter's elements.

<log spec> The <log spec> contains a logging code and, if logging is requested for an array, the ranges of the array to be logged.

<log code> The <log code> has the following meanings: 'I'—log as input to the asset; 'O'—log as output from the asset; 'X'—do not log.
The default format files are generated at compile time with the following defaults for parameter logging:

All scalar parameters are logged as I.
Scalar RETURN is logged as O.
Arrays of scalars are logged as I [(0 2) (-2 -1)].
Other parameters are logged as X (not logged).

A.2.4. Examples

Example: The compiler might produce the following three **`.fmt.dflt`** files.

`CubeSquare.fmt.dflt`

```
argc␣%d4␣X
argv␣%PR4␣X
^␣%v0␣X
```

No parameters are logged for the first asset in a program.

Square.fmt.dflt

```
x^%d4^I
y^%Pf8^I[(0^2)^(-2^-1)^]
^^%d4^O
```

The 4 byte integer x is logged on the asset call. The first three and the last two 8 byte doubles of the array y are logged on the asset call. The 4 byte integer return value is logged when the asset returns.

Cube.fmt.dflt

```
x^%d4^X
y^%Pf8^I[(0^2)^(-2^-1)^]
^^%d4^O
```

This is the same as **Square.fmt.dflt**.

Example: The Parameter Logging Editor in the interface might be used to create these **Square.fmt** and **Cube.fmt** files.

Square.fmt

```
x^%d4^I
y^%Pf8^I[(1^3)^5^(-3^-1)^]
y^%Pf8^O[10^]
^^%d4^O
```

The logging specification for y has changed. Seven elements in y are logged when the asset is called: y[1], y[2], y[3], y[5], and the last three elements in y. One element, y[10], is logged when the asset returns. The return value is still logged.

Cube.fmt

```
x^%d4^X
y^%Pf8^X
^^%d4^O
```

The parameters x and y are not logged. Only the return value is logged when the asset returns.

Appendix B

Unimplemented Features and Identified Problems

B.1. Unimplemented Features

Due to time constraints a number of features described in this thesis remain unimplemented at the time of this writing.

B.1.1. Parameter Logging Editor

- The **`validate_logging`** script has not been written for checking the compatibility between the **`.fmt`** and the **`.fmt.dflt`** files. When the program is recompiled, all the **`.fmt`** files are overwritten by the newly created **`.fmt.dflt`** files. Any previous editing of the **`.fmt`** files is lost.

B.1.2. Breakpointing

- Breakpointing conditions for parameters with types `char` and `char*` are not fully implemented. A `char` is treated as a number based on its ASCII code. Likewise, `char*` is treated as an array of ASCII values.

- There are no validity checks on the entries for Index or Value in the Breakpoint Browser. The Value field now only accepts numbers.

- Menus in the Breakpoint Browser are not context-sensitive. If a menu selection is not appropriate, nothing happens when it is selected.

- The menu selection Select all replicas and Deselect all are not implemented in the Breakpoint Browser list views.

- When a breakpoint is triggered, the identifying characteristics of the message responsible are not displayed in the System Transcript.

B.1.3. Sequential Debugging

- Currently *gdb* is the only sequential debugger that *Enterprise* attaches to assets during replay. Unlike most other public domain debuggers, it is compatible with C++, the language that the next version of the *Enterprise* executive is being written in.

B.2. Identified Problems

A number of problems became apparent during the development process. Again, due to time constraints and research priorities, solutions to the following problems have not been incorporated into the design description.

B.2.1. Animation during Replay

During program replay the animation system is always kept running. While this can be useful in analyzing message-passing behaviors and understanding the progress of the program, it is often undesirable due to the delays imposed by the animation. An event will not be replayed until the preceding event has completed animating. Therefore, it is desirable to introduce a switch in the interface menus to turn animation on or off during replay. This will eliminate most of the slow-down in execution speed during replay. Still, the additional overhead of sending replay instructions from the interface to the asset processes via the executive root process and sending acknowledgments back to the interface will continue to slowdown replay to an extent inversely proportional to the event granularity.

B.2.2. Communication Managers

A bizarre error deeply hidden in the NMP communication package consumed nearly a month of research effort. After the replay hooks were in place and working through the interface, an unusual problem developed. Regardless of what program was run from the Design View of the interface, the program would run correctly to completion, but the executive process would not exit. The code added to the executive to implement the replay facility should have had no effect upon the normal execution of a program. Soon it was discovered that if event logging was turned on, none of the events from Node 6 in the NMP process configuration appeared in the event file. It made no difference what the functionality of Node 6 was, the result was always the same. However, if the program was run from the command line using the same command generated by the interface, it ran and exited perfectly. This pointed to an interface problem, but none could be found. Moreover, if a program was run from the interface in replay mode, with all the additional message passing between the executive and the asset nodes, it once again ran and exited

perfectly. The problem was isolated to an anomaly in the pipe from Node 6 to Node 0, the executive, sometime after initialization. Node 6 got correctly initialized by a message from Node 0, and Node 0 received an acknowledgment of this from Node 6. Thereafter, though Node 6 continued to successfully *send* messages to Node 0 along an apparently intact pipe, Node 0 mysteriously never received them. Therefore, Node 0 would never record any events from Node 6 in the event file, and Node 0 would hang, waiting indefinitely for the Die message sent by Node 6.

After many unsuccessful attempts to further diagnose this problem, the replay hooks were re-coded, using another researcher's account, from the "replay development" version of the *Enterprise* executive into the then current version containing support for both PVM and NMP. When the re-coding was complete, everything worked perfectly—no problems with Node 0 losing contact with Node 6. Elation, however, did not last for long. When this researcher began working with the re-coded product in his own account, the same problem with Node 6 reoccurred! Inexplicably though, other researchers, logged into their own accounts but working in this researcher's directory with open file permissions, experienced no problems whatsoever. Conversely, when this researcher ran the new *Enterprise* code in another researcher's directory, the problem reappeared. It was then discovered that the *Enterprise* project account manifested a nearly identical error, but with Node 3, not Node 6, messages getting lost. The problem only occurred with NMP and only when run from these two user accounts. PVM worked flawlessly. This researcher then requested a new account with a different *uid* (user id), and the problem vanished.

The NMP developers suggested some fixes, but nothing succeeded in removing the problem from the *Enterprise* project account or this researcher's original account. It is suspected that an incompatibility exists with the currently used version of the Sun operating system (SunOS Release 4.1) and the NMP process management implementation. After this version of the operating system was installed, errors with NMP users having certain *uid*s had been reported. The incompatibility appears related to the fact that NMP processes are owned by `root`, not by the user. Why certain *uid*s caused particular pipes to breakdown during the execution of a specific version of the *Enterprise* executive, but only when run from the interface, was never determined. Whether the fault lies with NMP or with the operating system is an open question.

It is recommended, from this experience, that only PVM be used.

Lesson learned: Parallel distributed programs are indeed difficult to debug!

B.2.3. Persistence of Breakpoints

Currently, the expanded replicas of replicated assets are newly created as objects each time the user moves from the Design View to the Animation or Replay View. This

results in a lack of persistence in these Asset objects and their attributes (instance variables) whenever the user switches back to the Design View. Consequently, any set breakpoints that involve replicas will no longer be valid if the user switches out of the Replay View and back again. A mechanism for maintaining persistent replicas needs to be developed which allows replication factors to be modified in the Design View.

Moreover, persistence of breakpoint information needs to be extended to saved programs on disk, so that breakpoint settings need not be lost between user sessions.