INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Remote execution and debugging of computationally intensive programs in an inhomogeneous, distributed computing environment

Hahn, Paul Douglas, M.S.
Rice University, 1989

Copyright ©1989 by Hahn, Paul Douglas. All rights reserved.
RICE UNIVERSITY

Remote Execution and Debugging of Computationally Intensive Programs in an Inhomogeneous, Distributed Computing Environment

by

Paul Douglas Hahn

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Robert T. Hood
Associate Professor of Computer Science
Chairman

Hans-J. Boehm
Assistant Professor of Computer Science

Guy T. Almes
Assistant Professor of Computer Science

Houston, Texas
May, 1989
REMOTE EXECUTION AND DEBUGGING OF

COMPUTATIONALLY INTENSIVE PROGRAMS IN AN

INHOMOGENEOUS, DISTRIBUTED COMPUTING ENVIRONMENT

PAUL DOUGLAS HAHN

ABSTRACT

An ideal debugging system should provide the programmer with an interface to his parallel/distributed application that is consistent with that level of abstraction used in creating the application, while insulating him from differences in architecture, compiler, and operating system technology characterizing inhomogeneous computing environments. The nature of this abstraction problem is explored for various debugging systems, using a modeling method that interprets compilation and debugging as reciprocal mappings between adjoining levels of abstraction for the abstract data state of a program. Also, a new program (RXDM) is described that extends the capabilities of the existing debugger in the R^n programming environment to remote domains. Finally, a completely new system for execution and debugging is proposed, that will facilitate high-level debugging of parallel/distributed, optimized code in inhomogeneous environments. This new system uses a graduated hierarchy of abstraction layers, and a new intermediate language that provides primitives for communicating debugging information.
Acknowledgements

I wish to express my sincerest gratitude to my research advisor and thesis director, Robert Hood, for the many instructive, friendly, encouraging, and patient hours he generously invested in teaching and guiding me in connection with the work leading to this thesis. I also would like to acknowledge the other members of my thesis committee, Hans Boehm and Guy Almes, for their helpful comments on my thesis work. Furthermore, I wish to thank King Walters for having helped to make it possible for me to return to Rice in order to pursue an advanced degree, and thereby achieve one of my long-standing personal goals.

Finally, I of course would like to thank my wife, Tina, and my daughters, Rebekah and Miriam, to whom I am dedicating this work, for their loving support during these years of effort.

This work was supported in part by the National Science Foundation under Grant ASC-8518578 and by IBM Corporation through its research contracts with the Department of Computer Science at Rice University.
### Table of Contents

#### Chapter 1  Introduction ......................................................................................... 1

1.1 General .................................................................................................................. 1

1.2 Background and Motivation for Work ................................................................. 2

#### Chapter 2  Fundamentals of Debugging ................................................................. 6

2.1 Introduction ............................................................................................................ 6

2.2 The Input/Output Transformation Abstraction, Programming, and Debugging

2.2.1 Programs as Specifications of Input/Output Transformations .............. 7

2.2.2 The Empirical Nature of Debugging .............................................................. 14

2.2.3 Common Sources for Programming Errors ............................................... 17

2.2.4 Comparison of the Effort Involved in Debugging  ......................... 20
    Recursively Specified versus Non-Recursively Specified Transformations

2.2.5 The Empirical Debugging Cycle: An Iterative ................................. 23
    Convergence Mechanism to a Correct Transformation

2.3 Review of the Basic Elements of Debugging .................................................... 30

2.3.1 The Relationship Between Code Translation and ............................... 31
    Debugging; Translation Boundaries

2.3.2 Basic Elements of Execution Control in Debugging ............................ 37

2.3.3 Basic Elements of Data State Observation ............................................... 38
    and Modification

2.4 Special Issues in Parallel/Distributed Debugging ........................................... 40

2.4.1 Synchronization Conflicts and the Probe Effect ..................................... 40
2.4.2 Data Compatibility Conflicts in a Distributed, Inhomogeneous Computing Environment ........................................ 43

2.4.3 Debugging Equivalent Sequential and SMP Program Code via Reversible Execution ........................................... 49

2.4.3.1 Conditions for Reversibility of a Computation .......... 51

2.4.3.2 Implementation of Reversible Constructs ............... 53

2.5 Some Informal Comments on Debugging and Machine Learning (or, Man versus Machine) ........................................... 60

Chapter 3 The $R^n$ Model for Remote Execution and Debugging of Computationally Intensive Programs .................. 64

3.1 Introduction .............................................................................. 64

3.2 A Progression of Models for Program Execution and Debugging ................................................................. 64

3.2.1 Models for Local Program Execution and Debugging ................................................................................. 65

3.2.1.1 The Conventional Low-Level Machine Language Model ($adb$) .......................................................... 65

3.2.1.2 The Conventional Source-Level Model ($dbx$) ............. 67

3.2.1.3 The Basic $R^n$ Model .................................................. 70

3.2.2 Models for Remote Program Execution and Debugging ............................................................................. 71

3.2.2.1 The dbxtool Remote Source-Level Model .................... 74

3.2.2.2 The Extended $R^n$ Model, as Revised for Remote Domains .............................................................. 76

3.3 Implementing a System Based on the Remotely-Extended $R^n$ Model ............................................................. 79

Chapter 4 The RXDM Remote Execution and Debugging Monitor .......................................................... 82

4.1 Introduction .............................................................................. 82

4.2 General Description .................................................................. 83

4.2.1 Organization of the Monitor ............................................. 83

4.2.2 Monitor Directives: Commands and Service Requests ................................................................. 91
4.2.3 Detection and Reporting of Events: ................................................. 95
   Asynchronous Monitor Event Messages
4.2.4 Data Compatibility: Representation ............................................. 102
   of Typed Data Items in RXDM Messages
4.2.5 File System Interface and Special Monitor File Abstractions .......... 104
4.2.6 Controlling the Execution State of ............................................. 109
   Supervised Applications Programs
4.2.7 Controlling the Data State of ................................................... 114
   Supervised Applications Programs

4.3 Special Features .............................................................................. 117
4.3.1 Local Breakpoints Facility ......................................................... 117
4.3.2 Remote Program Environments and the .................................... 120
   Remote Function Call (RFC) Mechanism
4.3.3 Communications Between RXDM Monitors (talk Facility) ............ 123

4.4 Application of RXDM within the R^n Programming Environment .......... 127
4.4.1 The RXDM_FRIEND Remote Monitor Front-End .......................... 128
   Interface for Integrating RXDM with R^n's EXMON
4.4.2 Implementation Experience and Results ...................................... 131

Chapter 5 The Next Generation: An Advanced Abstract ...................... 133
   Debugging System for Conglomerate Computing

5.1 Introduction .................................................................................... 133
5.2 Criticism of the Remotely-Extended R^n Model for ............................. 133
   Program Execution and Debugging of Chapter 3
5.3 High-Level Abstract Debugging of ................................................. 138
   Distributed Programs: Objects and Handles
5.4 An "Idealized" Model for Remote ................................................... 142
   Program Execution and Debugging
5.4.1 Description of the General Model ........................................... 143

5.4.2 A Constrained Version of the General Model ............................. 151
    that is Suitable for Practical Implementation

5.4.3 A Canonical Intermediate/Intercomputer Transport .................... 153
    Language for the Constrained Version of the Model

5.5 Dynamic Program Restructuring via the .................................... 159
    Advanced Abstract Debugging System

5.5.1 Incremental Recompilation ..................................................... 160

5.5.2 Dynamic Insertion of Code Fragments into Existing .................... 163
    Program Text as a Specialized Debugging Technique

Chapter 6  Summary and Conclusions .............................................. 166

Bibliography ..................................................................................... 172

Appendix A  The Frequency Principle and Propagation of ...................... 178
    Errors in Tree-Structured Specifications

Appendix B  Solution to the Recurrence Relation for .......................... 183
    Propagation of Errors in Tree-Structured
    Specifications
Chapter 1
Introduction

1.1 General

The efficient and effective debugging of parallel/distributed programs is an issue of increasing interest and concern in the computer science community, both in the industrial and academic realms. This is especially the case as hardware systems suitable for medium- to large-scale parallel/distributed applications have become both more powerful as well as less costly, and therefore more accessible to a more extensive and enthusiastic retinue of software developers than ever before. Simultaneously, parallel/distributed programs developed on and for these systems have become larger and more complex. This has resulted in the corresponding phenomenon that efficient development of error-free, very large scale parallel programs has become increasingly difficult, being exacerbated to a large degree by the compounding of subtle forms of errors ('bugs') that are specifically associated with parallel execution, such as errors connected with the timing or synchronization of the various parallel tasks involved.

In fact, the debugging process is perhaps the most labor-intensive part of any large-scale parallel programming effort. In order to meet this challenge, more powerful parallel program development systems, or programming environments, have been — and are being — designed and implemented, and are supplied with special features related to the debugging of such large and complex applications programs. Depending on the system, these features might include facilities aimed at providing interactive support to the programmer/user for supervised, semi-automatic program semantic analysis at program specification-time, as well as high-level debugging functions during program run-time. One such parallel programming environment is that being developed under the auspices of the $\text{R}^n$ project currently underway at Rice [CCH+87].

This thesis is concerned principally with addressing the issues related to the effective and efficient run-time debugging of large-scale (i.e., 'computationally intensive') parallel programs operating in a distributed computing environment. In particular, I first present a general discussion of debugging and some of the related issues involved therewith in Chapter 2. Next, in Chapter 3, I develop a computational
model for debugging in a distributed computing environment, that is oriented specifically for $\mathbb{R}^n$. This model then is used as the basis for implementation of a remote execution and debugging monitor called RXDM, basically for experimental use in the $\mathbb{R}^n$ project, which I describe in Chapter 4. Finally, in Chapter 5 I explore the limitations of the $\mathbb{R}^n$ debugging system, as extended for remote operation via RXDM, and then, as a consequence of this, propose a more advanced, abstract debugging system that is based upon a more complete, high-level reformulation of the computational model for debugging in distributed computing environments, including dealing with optimized code.

1.2 Background and Motivation for Work

The concrete mechanics involved in the debugging process for parallel programs in distributed computing environments in particular, can be complicated by factors related to installation-dependent differences in the configuration of such environments. For example, the distributed environment might consist of an inhomogeneous mix of machine architectures. Also, the various code segments residing throughout the distributed environment that constitute a given parallel/distributed program may have been generated using different compilers, targeted either for the same or different programming languages, and even may be running under different operating systems.

In any case, the fundamental concern is how to deal with disparate representations of debugging-oriented information that can result in and from such a generally inhomogeneous situation. For example, as a result of architectural and/or compiler differences, the representations of certain 'computational objects' making up the abstract data state of a parallel/distributed program can differ from one parallel code segment to the next, perhaps in a fundamentally incompatible way, such that efficient and meaningful communication and interpretation of these objects throughout the distributed computing environment becomes a complicated issue, strongly sensitive to variations in the installation-dependent configuration of that environment. Note that these computational objects might range from instances of the 'primitive data types' such as integers, floating point numbers, address pointers, etc., to more complex and abstract program objects such as call frames on the program run-time stack. For example, with respect to the former, the corresponding
representations may be incompatible due to architectural differences in regards to byte order, word size (which relates to precision), sign representation, etc. For the latter type of higher-order computational objects, the corresponding representations may be incompatible due to the format or structure, and/or definition of the individual information items constituting the object, which is often dictated as much by the given compiler design (and, quite possibly, the associated, target programming language), as by any architectural considerations.

Furthermore, the problem can be generally compounded by any operating system differences extant in the distributed computing environment. In particular, different operating systems may provide differing mechanisms for accessing debugging-oriented information relating to a program, as well as explicitly controlling the execution of that program. Also, different operating systems may incorporate differing interprocess communications paradigms (for example, asynchronous, signal/interrupt-driven message handling is available in UNIX\(^1\) systems, but not in V (reference [Che88] and [CZ83]), which can affect the communication of debugging-oriented information between computers in the environment, both from the standpoint of communications efficiency as well as effective interpretation of the information. Thus, at least the design of the network interface portion of the distributed debugging system will be strongly influenced by operating system characteristics.

Basically, dealing with this general problem is an exercise in constructing an appropriate abstraction scheme. That is, the fundamental goal is to insulate the programmer and the 'local' portion of the debugging system\(^2\) from configuration-dependent variations in the representations of the debugging-oriented information that can occur throughout the distributed computing environment. Various debugging schemes or models therefore result from this common motivation, where the differences between one such scheme and another largely have to do with the particular level at which the corresponding abstracting away of architecture- and compiler-, and operating system-dependencies is effected.

---

\(^{1}\) UNIX is a registered trademark of AT&T Bell Laboratories.

\(^{2}\) i.e., that portion with which the programmer interacts
In particular, one well-known approach for dealing with this problem is to use a locally-based, overlying interface layer to operate in conjunction with individual, remotely-based instances of a standalone debugging system. (For example, this is the scheme for dbxtool [AD86], where the targeted standalone debugging system is dbx [UPM83].) The standalone debugging system normally would be used in a non-remote scenario, and therefore is itself the prime element effecting execution control and debugging of the individual code segments under its supervision. The interface layer, on the other hand, is designed to function basically as a communications hub or controller, interfacing between the locally-situated programmer and the remote instances of the standalone debugging system, that facilitates transportation of debugging-oriented information across the network between these two agents. The critical feature of this sort of scheme is that the debugging information so communicated is at a relatively high level of abstraction; that is, at least at the same level as in the non-remote situation where the programmer interfaces directly with the standalone debugging system. This then accomplishes the aforementioned goal viz. abstracting away of all architecture-, compiler-, and operating system-dependencies. Nevertheless, a relatively high price is paid in terms of adaptability of the debugging system to different configurations of the distributed computing environment, since each instance of the standalone debugging system must be explicitly designed for a specific combination of architecture, operating system, and compiler/programming language. Also, the scheme is space-inefficient, since there must be an individual, complete instance of the standalone debugging system associated with each parallel code segment residing throughout the distributed environment. In addition, the scheme in some cases can be time-inefficient as well, since the communication of all debugging information is carried out in a symbolic format, involving possibly lengthy variable names, etc.

These sorts of problems consequently motivate the design of a new remote debugging system for application in an inhomogeneous distributed computing environment. In Chapters 3 and 4 of this thesis, I present a description of one such approach that has been implemented for the Rice R^3 project, in which the existing standalone debugging system, known as EXMON (from EXecution MONitor), was extended for remote operation via the addition of a new underlying abstractions/communications layer (reference the aforementioned RXDM Remote eXecution and Debugging Monitor) between it (EXMON) and the various remotely distributed programs/program code segments. This approach, which involves only a
single, locally-based instance of the original standalone debugging system, stands in contrast to that described above for dbxtool, since the latter involved the addition of a new overlying interface layer between the (local) programmer and the manifold of remotely distributed instances of the original standalone debugging system, dbx.

Nevertheless, as it turns out, this new approach also has its own shortcomings with respect to some of the aforementioned problems. Consequently, I present in Chapter 5, for the purpose of future research, specifications for the design of what might be termed an 'idealized' debugging system, which ought to be free of all such problems.
Chapter 2
Fundamentals of Debugging

2.1 Introduction

In this chapter, I will attempt to discuss what debugging is all about. Although this may at first appear a trivial topic for the casual reader, I attempt herein to explore the subject on a more abstract and conceptual level. First, I introduce the concept of input/output transformations as a foundation for viewing the programming and debugging tasks in a more conceptual and mechanistic light. Next, I explore some of the basic elements of debugging, as applied both to sequential and parallel/distributed programming, and then discuss some special issues in parallel/distributed debugging. Finally, as an informal caveat, I attempt to show that the programming/debugging process is fundamentally interrelated and interwoven with the 'machine learning cycle', and thereby establish an analogy between the programming/debugging process and the functionality of machine intelligence.

2.2 The Input/Output Transformation Abstraction, Programming, and Debugging

Obviously, debugging is made necessary by the fact that 'errors' can occur in a program, being that activity or process by which such errors are corrected. But what are errors? And, for that matter, just what is a program? Well, without attempting to insult the reader's intelligence, a 'program' is simply the result of 'programming', which can be defined as that activity or process corresponding to the specification of an 'input/output' relationship or transformation (which can also involve temporal aspects) to a machine, for the purpose of exploiting that machine to aid in some application, such as solving an abstract problem, or to perform a control-related task. This transformation is thereafter carried out by the machine, acting on its inputs (which may be vacuous) to yield outputs useful to the user for the application at hand.
The goal of this section will be to discuss the nature of programming and debugging in the context of this input/output transformation abstraction.

2.2.1 Programs as Specifications of Input/Output Transformations

Programming requires, by its very nature, the ability on the part of the user to provide an accurate specification of the input/output transformation in a compact, summary form. The fewer number of 'bits' required to provide a complete and accurate specification of the transformation, the better off is the programmer's lot in life (of course, this is basically what drives programming language research). The converse of this would be that, if possible, the user specify the entire transformation by means of complete exhaustion of every possible input/output pair relation.\(^1\) Note that, this is effectively what is done in programming state machines, as for digital control applications: namely, a sequence of statements of the form

"if (simple logical relationship on two inputs is true) then (outputs)"

where the two 'inputs' can either be externally-supplied arguments, or 'outputs' from a previous statement, and the 'simple logical relationship' is either and or or, where possibly each can be negated with not. However, this exhaustive form of specification is not always possible, because the actual number of input/output pair combinations is usually unmanagably large, even infinite in extent. Therefore, the user is usually motivated to specify the input/output transformation in compact (that is, finite) form by means of higher-level abstractions. For example, a differential equation expression such as \(\frac{dy}{dx} + ky = 0\) is simply a very compact, highly abstract form of specification for the following (infinite) set of input/output pairs (the solution set):

\[ \{ (x, y) \mid Domain(x) = \text{Reals} \land y = e^{-kx} \}. \]

---

\(^1\) including temporal behavior, as with sequences
Note that the abstract form of specification (in this case, the differential equation) is always a finite representation of the input/output transformation (e.g., the mapping of input $x$ to output $y = e^{-kx}$), whose measure, in terms of the number of possible input/output pairs, is infinite. That is in fact what symbolic representation is all about; namely, the means to specify an input/output transformation, with possibly infinite measure, in terms of a finite number of abstractions, or symbols.

Of course, there are many alternate ways to specify the same transformation between $x$ (input) and $y$ (output) in this (or any) example, mostly dependent upon the available relative level of abstraction naturally expressible in the programming language being used as a substrate for interaction with the machine. That is, there are languages that admit a very high level of abstraction expressibility — in fact, in this example, there are languages that admit the direct and literal expression of the differential equation itself as an acceptable specification for the indicated input/output transformation. On the opposite extreme, there is the native machine code language for the underlying machine architecture, which provides the lowest possible level of abstraction expressibility for specification of input/output transformations — in this example, the specification would necessarily involve some embedding of the axioms of differential calculus, as complicated sequences of instructions, in order to enrich the available language constructs sufficiently enough so as to permit formation of a concrete representation for the input/output transformation in the machine. Thus, the level of abstraction expressible in a particular language corresponds to the power of the axioms already embedded in the language, at least in terms of the suitability of such axioms for a particular application.

Note that any specification for an overall transformation is always composed from sub-specifications for lower-level (viz. meaning or functionality) component transformations operating with and upon certain specific data structures. These sub-specifications are in turn themselves composed from sub-specifications for even lower-level transformations; thus, the overall specification is really a recursive hierarchy of sub-specification, and likewise for the equivalent input/output transformation. This is illustrated in Figure 2.1. Arrows in the figures represent generalized data and/or control dependencies. Note that the 'base case' specification in this recursive hierarchy, corresponding to the case having the lowest level of abstraction, simply could be a primitive arithmetic machine code instruction with (say) symbolic operands. Likewise, the corresponding 'base case' input/output transformation would be the equivalent concrete machine operation that performs the specified primitive
operation on its 'inputs' (operands), and produces some 'output' value (result operand).

(a) Hierarchical, Abstract Specification

(b) Corresponding Hierarchical, Abstract Input/Output Transformation

Figure 2.1: Hierarchical Nature of Abstract Specifications and Corresponding Input/Output Transformations

I will now attempt to compare the relative effect of errors occurring in specifications that differ in terms of their level of abstraction, but that represent the same input/output transformation. First, I adopt the following principles:

(1) In general, the probability of occurrence of at least a (fixed) given number of errors in a specification is related to the length of the specification, increasing as the length increases; furthermore, the overall frequency of occurrence, or density, of errors in a specification is
related to the length of the specification, increasing as the length increases (the \textit{frequency principle}). This principle applies primarily to the occurrence of semantic (meaning) errors in a specification; syntax errors probably will be randomly distributed throughout the specification, with a \textit{constant} density with respect to the length. I believe that this principle can be largely attributed to what could be termed the 'confusion factor' — namely, as the length of a specification increases, it becomes, in general, more and more 'confusing' to the programmer, both from the standpoint of development as well as maintenance. For example, the longer the specification, the more likely it will be for the programmer to omit details, making the specification \textit{incomplete} \textit{viz.} the envisioned, 'correct' input/output transformation. [Trivial Corollary: The probability that a specification is 'correct' decreases rapidly as the length of the specification increases. Likewise, the \textit{expected} number of errors in a specification generally increases at a more than linear rate as the length increases.]

Note that Appendix 1 provides further discussion of this principle, as well as some illustrative examples.

\begin{enumerate}
\item In general, the probability of a significant \textit{manifestation} of any type of error in a specification, as reflected in the likelihood of appearance of some sort of evidence for the error in any fixed set of experimental observations on sample input/output pairs, is related to the \textit{length} and the \textit{semantic complexity} of the specification (\textit{i.e.}, complexity in terms of the 'extent of meaning' of the various abstractions or symbols actually used in forming the specification), decreasing as the length increases, and increasing as the complexity increases (the \textit{manifestation} or \textit{subtlety principle}). Thus this principle embodies the effects of two competing forces, one involving a tendency to decrease the observable effects of an error with an increase of the length of the specification in which the error is embedded (error \textit{dilution}), and the other involving a tendency to increase the observable effects of an error with an increase of the structural (semantic) complexity of the specification (error \textit{propagation}). A major factor contributing to the latter is that any specification for the overall transformation is always constructed using less semantically-complex sub-specifications for \textit{component} transforma-
tions. Consequently, when the individual component transformations are combined, any errors (syntactic or semantic) contained in the individual components can be magnified in terms of their net effect on the observed input/output behavior of the overall transformation, thanks to the propagation of the effects of the errors via the sharing of data between the components (i.e., the 'whole is greater than the sum' with respect to the observable effects of program error propagation).

Note that this principle relates to the probability of observing the effect(s) of an error in a specification, whereas the frequency principle relates to the probability of there even being a (given) number of errors in a specification.

(3) The relative impact of any type of error in a specification, in terms of the magnitude or severity of deviation of the observed from the expected outputs (taken over all possible inputs), is related to the semantic complexity of the specification, increasing, in general, as the complexity increases (the impact principle). Note that this principle could have associated with it the simple error metric $E_D^2$, where

$$E_D^2 = \sum_{i \in D} (o_{\text{exp}}(i) - o_{\text{obs}}(i))^2,$$

where $o_{\text{exp}}(i)$ and $o_{\text{obs}}(i)$ are the expected and observed outputs for the input value $i$ in domain $D$, respectively. A pathological case viz. the impact of an error would be when a certain input value induces an error that results in a trap exception on the machine (e.g., a segmentation fault as the result of an attempted illegal pointer reference). In this case, the corresponding, observed output value will be undefined, so the calculation of the error metric will also be undefined.

Note that this principle does not relate to probabilities, as in the above two instances. In particular, it differs from the subtlety principle in

\[\text{footnote}^2\] Of course, this discrete sum may have to be replaced by an integral over the inputs domain, in case the domain is piecewise continuous instead of discrete.
that the latter refers to the likelihood of being able to observe the
effects of an error in any fixed-size sample of the outputs, whereas this
principle concerns the total effects of all errors, irrespective of their ex-
perimental 'observability'. Nevertheless, it is this principle that really
forms the basis for all 'empirical' debugging — namely, that the pro-
grammer attempt to zero out the above (or, some) error metric via re-
petitions of a cycle involving experimental observations upon and
modifications to the concrete input/output transformation. This pro-
cess is discussed in greater detail in following sections.

Now, as alluded to, all three of the above principles are interrelated in one way or
another. Part of this interrelationship is due to the following caveat: the degree of se-
matic complexity of a specification generally is inversely related to the length of the
specification, such that if the specification for a given (fixed) transformation is made
less semantically complex (e.g., is formulated using less abstract constructs), then its
length almost invariably increases.³

However, the converse relationship is not so tight, in that there exist textual
transformations that can affect the length of a specification without affecting either
its semantic complexity or overall meaning. For example, one can artificially
increase/decrease the length of a specification by the arbitrary addition/deletion of
identity relations (such as statements of the form "a = a"), without affecting the lev-
el of semantic complexity inherent in the original specification; the new specification
will still correspond to exactly the same input/output transformation. (Note that the
elimination of extraneous identity relations from a specification is one type of optimi-
ization.) Nevertheless, it is a generally valid statement that, in order to decrease the
length of a specification for a given input/output transformation, higher levels of
abstraction must be utilized, corresponding to to an increase in the meaning content
of the (fewer number of) symbols used in the representation (i.e., an increase of the

³ For example, note that a subroutine call is considered more semantically complex than the
equivalent text sequence it replaces — likewise, the length of the subroutine call construct is almost al-
ways much less than that for the equivalent text sequence. Another example would be the situation in
which a high-level specification is reformulated using a different language having less abstraction ex-
pressibility, such as the assembly language for the target machine.
semantic complexity of the specification). Note that it is difficult to come up with a good, generally-applicable metric for the degree of semantic complexity of a specification, as was done in the case of $E_D$ for the impact principle above. This is possibly a matter for further research.

Now, according to this, a more highly abstract form of specification for a transformation will in general be shorter (that is, more compact) than a less abstract form of specification for the same transformation — therefore, an error will be less likely (per the frequency principle) for the highly abstract form, but probably will be more profound in terms of the likelihood of observing its effects (per the subtlety principle) as well as the relative magnitude of these effects (per the impact principle). Now, since the natural tendency is to specify a transformation using the highest possible level of abstractions available in order to facilitate program development as well as subsequent maintenance (i.e., ease of implementation of changes), errors at such a high level of abstraction will be less likely, but typically more profound in their manifestations *viz.* the performance of the program. The following conclusions may be drawn from this observation:

- Errors in highly abstract specifications are easier to detect as a consequence of the decreased subtlety of their effects, and *vice-versa* for less abstract specifications for the *same* transformation.

- In contrast to the above point, although errors resulting from maintenance changes to highly abstract specifications are probably less likely, they probably will be more profound (in other words, the more abstract the specification, the easier it is to break or corrupt the associated input/output transformation in a significant way as the result of committing a relatively 'small' mistake). One could therefore say that less abstract specifications are, overall, more impervious to the effects of 'mistakes' than are equivalent highly abstract specifications.

- Just as the effects of errors in less abstract specifications are more difficult to observe (due to the subtlety principle), the effects of *attempted corrections* to such errors will likewise be difficult to observe. Therefore, in general, empirical debugging (as discussed in the following sections) of less abstract specifications is made doubly difficult — that is, not only in terms of the detection and location of an error, but also in terms of fixing it.
2.2.2 The Empirical Nature of Debugging

In the context described above, program errors are simply deviations from the ‘correct’ input/output transformation as envisioned by the user (purposeful emphasis). In other words, from the usual semantic point of view, the meaning of the program is wrong. Likewise, debugging is the process of modification of an implemented input/output transformation (rather, its specification) in order to bring it into conformance with the envisioned, correct transformation. Pragmatically, debugging cannot be conducted purely on the basis of formal, symbolic specification methods. Instead, the debugging activity is usually highly empirical, and is carried out — using the appropriate tools — just far enough to provide sufficient evidence to a programmer/user that his program ‘works’. (This last term is used in order to emphasize that, whether a program functions correctly or not is, in the real world, frequently left as a matter of human perception. In fact, in light of the observation that even the ‘correct’ input/output transformation for a particular application is ultimately a product of the user’s visualization, taking into account an analysis or perception of the application itself, then one might say that the entire programming process, including debugging, is actually completely dependent upon human perception.)

Usually, this evidence is in the form of sets of experimental input/output combinations, perhaps intentionally formulated according to some verification guidelines so as to provide a statistical confidence interval measurement of the expected ‘correctness’ of the program that is acceptable to the user. Of course, if it is known that the total number of such unique combinations is finite, then it is theoretically possible to experimentally verify all the combinations, but this is frequently impractical for all but the most straightforward programs, thanks to the usually immense number of such combinations. Furthermore, for this finite case, if one knew explicitly all the inputs/outputs combinations beforehand, then it may be questionable for some applications as to whether it was even necessary to create the input/output transformation on a machine in the first place. Obviously, although this is the case for automated discrete digital control applications, as well as for some database applications, the answer to this rhetorical question is nevertheless ‘YES’, since the goal here is to take advantage of the speed and convenience provided by the machine in accessing and manipulating the stored information.
On the other hand, most real-world input/output transformations are designed for operation with an inputs domain that is open-ended, so the number of input/output combinations is infinite. In fact, such an infinite domain may or may not even be countable in the mathematical sense. Furthermore, even if the application-specific inputs domain is bounded, the actual computational domain on which the input/output transformation is defined (i.e., will produce a valid output for any given input from the domain) may still be infinite in measure.

This is basically because of two reasons. First, the bounded application domain may not be countable (e.g., real numbers between an upper and a lower limit, say 0.0 and 1.0), so the number of input possibilities (viz. the actual computational domain) is infinite. Secondly (but not necessarily exclusive from the first reason), although the application domain may be bounded and countable, therefore itself yielding a finite number of input possibilities, the actual computational domain may be unbounded, since it is the same as the output domain the entity that physically generates the inputs. For example, a program may have to operate on the finite application domain in the normal case, but in addition must produce some sort of predefined response(s) (e.g., ‘error output’, being perhaps a flag or special signal) for any other input that can be generated but is not specifically in the application domain — that is, the program must ‘reject’ possible inputs not in the application domain. In other words, from the standpoint of robustness, a program ideally must be flexible enough to perform, in a deterministic and reasonable manner, some sort of ‘real-time debugging’ of the external environmental conditions under which it is running. This includes, among other things, domain checking of the inputs provided it. A concrete example would be the function ‘\( a \mod b \)’, which is an input/output transformation between the (input) numbers \( a, b \) and the (output) number that is the remainder of the integer division of \( a \) by \( b \), where \( a \in \text{Integers} \cap [0..32767] \) and \( b \in \text{Integers} \cap [1..32767] \) is intended to be the application domain, which is bounded and countable, and therefore finite. However, if the inputs are taken from a user at a terminal, then the worst-case computational input domain is really all numbers expressible from the terminal and on the machine running the program, since a naïve or careless user has the capability to press any key, thereby inputting just about anything (even non-numeric characters). Therefore, the input/output transformation (program) implemented for this function should be designed robustly enough to deal with input from the computational domain that is outside this nominal application domain. In particular, at the programming language statement level of abstraction,
the input statements may permit the programmer to specify that input values be restricted only to integers; also, only variables having the type integer could be used to ensure the restriction of the domain. In addition, at a higher level of semantic abstraction, the function might be defined to produce the distinguished output ‘−1’ (say) for \( b \in \text{Nonpositive Integers} \).

In any case, the correctness of such a program cannot be verified purely on the basis of experimentally observing all possible input/output combinations in the computational domain; certainly, even an attempt to do so just for all the (finite number of) combinations involving inputs taken from the original application domain would be insufficient. Note, however, domain checks (for example, in the form of assertions) included in a program are often quite straightforward, and therefore individually amenable to direct, formal proof methods (although formal type analysis is not always so straightforward), so these formal (symbolic) methods can often be successfully combined with the empirical input/output observations in debugging such programs (reference [Cla75] and [Sma79], as well as the discussion in section 2.2.5).

Note that, even for those applications that involve a finite computational domain, such as automated discrete digital control (where the computational domain is all possible combinations of a finite number of discrete input values), the finite computational domain typically will be (much) larger in measure than the intended application domain. In this example, for instance, the intended application domain typically is a small number of ‘legal’ input (switch) combinations, in comparison to the relatively large number of possible input combinations constituting the computational domain. Consequently, initial passes at implementing digital control programs are frequently faulty with respect to handling unanticipated input combinations, even though they are verified to produce a correct output for every ‘legal’ input in the application domain. Naturally, this is a matter of grave concern for users of these programs, where such a faulty control program could cause an unexpected and expensive shutdown, or worse, of a manufacturing plant (say), as the result of an intermittent switch closure or wiring short generating an unanticipated input in combination with an already-present, ‘legal’ set of inputs.\(^4\)

\(^4\) A software-engineering related comment: Based on my own experience with the development and debugging of control programs, I feel that it is somewhat surprising, and even alarming, how frequently little or no provision is made in typical control programs for robust handling of unanticipated input
In summary, therefore, one can see that, for transformations defined on computational domains with infinite (or even large, finite) measure, one often needs to incorporate statistically optimized experimental observations on input/output combinations with other, non-empirical debugging methods, in order to achieve a given required level of confidence (expressed as a probability) in the 'correctness' of the program. For some applications, this required level of confidence should in fact ideally approach 100% for all inputs in the entire computational domain, as illustrated with the above example viz. discrete digital control. However, it will not be a goal of this thesis to enter into discussions of formal proof methods, nor the explicit statistical considerations involved in formulating an optimized approach to empirical debugging (see, for example, the discussion of testing theory by Hamlet [Ham84]), but rather some of the mechanics and philosophies directly involved in the empirical debugging process.

2.2.3 Common Sources for Programming Errors

In this section, I will discuss some of the common sources for errors in programs from the point of view of input/output transformations. Once again, program errors are simply deviations from the 'correct' input/output transformation as envisioned by the programmer/user. That is, from the usual semantic point of view, the meaning of the program is wrong. Error deviations arise from two sources: namely, 'external conditions' and 'internal bugs'. The former includes those real-world parameters that are combinations from outside of the application domain but within the worst-case computational domain. In fact, the debugging process for real-world control programs typically involves exhaustive empirical verification of the input/output combinations only for all inputs occurring in the application domain, and perhaps a few others from among the remaining possibilities. Of course, a large total number of possible input combinations in the computational domain sometimes makes a complete verification effort by this exhaustive technique impractical due to schedule and/or cost constraints. In these situations, an attempt is sometimes made to verify correct (or reasonable) behavior by the program with a limited set of 'illegal' input combinations not in the application domain, by first predicting which such input combinations might otherwise possibly result in some worst-case, catastrophic behavior. Based on experience, however, it is unfortunate that the list of worst-case illegal input combinations so investigated is sometimes disastrously incomplete.
outside of the explicit program development loop, such as hardware failures (soft or hard) or system software bugs (operating system, compiler, etc.). These parameters are often spurious and unpredictable, and, likewise, frequently outside of the control of the user, who consequently must (rather, ought to) attempt to deal with them on a robust but nevertheless stochastic basis.

Internal program bugs, on the other hand, can be the result of two possibilities, both under the programmer's control. The first is that the programmer did not even envision a correct or consistent input/output transformation in the first place. In other words, the particular algorithm selected by the programmer may not be appropriate for the application at hand. For example, this can be the case especially if incorrect assumptions are made concerning the model for the application (e.g., the use of the wrong physical laws as applied in the calculation of a trajectory), or if the algorithm involves any type of heuristics that might be inappropriate or incomplete with respect to the application (e.g., the choice of inappropriate error bounds as a stopping criterion for an iterative numerical calculation). I shall term this class of bugs 'logical abstraction' errors (or simply abstraction errors), since the error is invariant in regards to how the transformation is subsequently specified to the machine; i.e., the error is independent of implementation details, or rather, any explicit programming activity on the part of the programmer. Thus, abstraction errors occur in the 'on-paper' design of the algorithm, and are conceptual in nature.\(^5\)

The second possibility is, assuming that the transformation envisioned by the programmer is sound, the manner in which the transformation was specified to the machine (viz. the programming activity itself) may be erroneous. This includes both program 'syntax', and 'semantic' errors. Syntax errors correspond to a failure on the part of the programmer to communicate accurately with the machine. Insofar as their occurrence in a specification is concerned, they may occur either randomly (i.e., typographical errors in a program), or systematically (i.e., misuse of programming language constructs, such as mismatch of typed data operands in a type-restrictive operation). Syntax errors are usually detected automatically and reported by the machine (i.e., the compiler/interpreter) during translation. Semantics errors,

---

\(^5\) See also the definition of 'misconceptions' given by Joni [JSG+83].
on the other hand, correspond to a failure to construct, at some relatively low level of abstraction, a proper machine-interpretable specification (program) whose meaning, as interpreted by the machine, corresponds to the overall transformation to be implemented. As touched upon in a preceding section, this type of 'program meaning' error often becomes more likely in proportion to the complexity of the algorithm, and consequently in proportion to the 'size' or length of its specification (in accordance with the frequency principle), such that the total number of errors in a specification will generally increase with an increase in the length of the specification more rapidly than in a simply linear fashion (refer to section 2.2.1).

Therefore, even small decreases in the length of a specification can often be effective in reducing the possibility of errors. Typically, this length decrease can be accomplished by increasing the semantic complexity of the specification. That is, any specification for an overall transformation is always composed from sub-specifications for lower-level (viz. meaning or functionality) component transformations. These sub-specifications, in turn, may be themselves composed from sub-specifications for even lower-level transformations; thus, the overall transformation is really a hierarchy of sub-transformations (see Figure 2.1), and this is reflected in the structure of the corresponding concrete specifications. An increase in semantic complexity of a specification for a given overall transformation is therefore accomplished by constructing it from more highly abstracted sub-specifications corresponding to more highly functional component transformations. In concrete terms, this basically means the use in the specification of symbols with greater 'extent of meaning'. The use of function or subroutine references to represent condensations of duplicated or highly similar blocks of code would be one simple example of this. Another example would be to reformulate specification in terms of more abstract operations and associated data structures that more appropriately fit the application. Similarly, one could use a different programming language having more powerful constructs, in the sense that the axioms embedded in the language are more appropriate for the application.

On the other hand, increasing the semantic complexity of a specification for a given transformation can sometimes result in increasing the magnitude of the observed effects of the (albeit diminished number of) errors in any sample execution of the transformation (in accordance with the impact principle). In other words, the greater the semantic complexity of a specification for a given transformation, the more sensitive it will be to 'small' errors in terms of their net impact on the overall performance of the transformation. This is especially relevant with respect to the
program maintenance effort for many large software products, which typically entails well over half the total costs (viz. manpower expenditure) for the product. Actually, though, this is a positive attribute from the standpoint of debugging the specification, since it implies that the detection of errors and their subsequent localization and correction becomes effectively more straightforward as the semantic complexity of the specification is increased.\(^6\)

Therefore, to reiterate, semantic errors, in contrast to abstraction errors, are connected with the particular implementation chosen for the overall transformation or algorithm. Abstraction errors, on the other hand, are conceptual errors in the algorithm itself, and are unrelated to the physical programming effort. Note, though, that the terminology is relative in the sense that each component transformation, if viewed as an individual object, also could itself contain both abstraction errors (viz. conceptualization or functionality), as well as semantic errors (viz. implementation).

2.2.4 Comparison of the Effort Involved in Debugging Recursively Specified versus Non-Recursively Specified Transformations

Insofar as debugging is concerned, the recursive method of specification of a transformation, when possible, can be advantageous in some situations from the standpoint of lowering the propensity for errors in the specification, at least in comparison to equivalent non-recursive specifications for the same transformation. In other words, if a transformation admits equivalent recursive and non-recursive implementations, and assuming that no previous debugging has been performed on either form of implementation, then debugging the non-recursive form is frequently at least as hard as debugging the recursive form. The reason for this is that, in most cases, the number of component transformations involved in the non-recursive form of specification is at least as many as the number involved in the recursive form of specification, since the latter is self-referential.\(^7\) Furthermore, the probability that a semantic (or even syn-

---

\(^6\) This is one of the reasons for advocating the 'structured programming' technique.

\(^7\) This is consistent with the popular observation that the recursive form of specification is frequently shorter (lengthwise) than an equivalent non-recursive specification for the same transformation. Likewise, the recursive form of specification, which is self-referential, typically contains fewer unique sub-specifications than a non-recursive equivalent.
tax) error will be present in the overall specification is proportional to the number of unique sub-specifications of component transformations appearing in the overall specification (again, in accordance with the frequency principle).

On the other hand, the non-recursive implementation will involve sub-specifications for component transformations that can be separately debugged, and this really needs to be done only once for each, since the resulting 'correct' versions could be stored away in a library for later export and use. So a non-recursive specification made up from these 'correct' sub-specifications for the component transformations would enjoy the benefit of the previously-applied debugging efforts, whereas the recursive specification will almost always have to be debugged from first principles in each instance. Nevertheless, if debugging of the recursive and non-recursive cases is applied from the same starting point — i.e., in the sense that no amount of debugging on any of the components was previously carried out for either case — then the recursive case should in general have a lower propensity for errors than the non-recursive case. Consequently, the conclusion is that programming in the recursive style, though perhaps yielding a less efficient transformation viz. execution time/space (typically due to the additional overhead involved in a deeper sequence of function calls), is potentially superior in terms of the required effort in debugging and maintenance.

Note that, in the recursive case, the specification for the overall transformation involves identical sub-specifications of the same transformation, only operating on reduced sets of inputs. Since the specification of the transformation is therefore self-referential, then if the overall specification is correct, then the sub-specifications of the same transformation must trivially be correct as well. Likewise, if one assumes that the recursive sub-specifications are correct, at least when applied to the reduced inputs, then it is sometimes a straightforward excercise to generate an inductive proof of the correctness of the overall transformation.8

A ubiquitous but simple example that illustrates the comparison between the recursive and non-recursive cases is the specification of the factorial function (an

8 Note that, conceptually at least, debugging is, in general, sort of a real-time excercise in generating at least a partially complete correctness proof for a transformation.
input/output transformation between the integer \( n \) and the integer \( \prod_{i=1}^{n} i \). The usual way to recursively specify this transformation is the set

\[
\{(i, f(i)) \mid i \geq 0, \ f(0) = 1, \text{ and } \forall i \geq 1: \ f(i) = i \cdot f(i-1)\}
\]

Also, one (structurally very similar) way to non-recursively specify the same transformation is the set

\[
\{(i, f(i)) \mid i \geq 0, \ f(0) = 1, \text{ and } \forall i \geq 1: \ f(i) = i \cdot \text{prod}(1, i-1)\}
\]

where

\[
\text{prod}(n, m) = \prod_{j=n}^{m} j.
\]

Both cases involve similar sub-specifications of component transformations, although the recursive case involves the sub-specification of the identical transformation, operating on a reduced input. Thus, debugging of this case is a sort of 'bootstrapping' operation (i.e., dynamic implementation of the usual static, induction-based proof process); if the lower-level transformation is debugged and consequently correct for all inputs up to and including \((i-1)\), then (inductively) so is the overall transformation if the subtext "\(i \cdot f\)" is debugged and correct. The iterative case, however, involves an additional, separate sub-specification for the 'product' function transformation, possibly in the form of a separately-compiled subprogram. Consequently, this could be a separate source for errors altogether, and its correctness must be separately established. However, as alluded to above, once debugged and certified 'correct', the product function \(\text{prod}()\) could be stored away in a library for later use as a component transformation in other applications as well, without again having to undergo any debugging. Therefore, one can see from this simple example how, if both the recursive and non-recursive forms of a specification are developed and debugged from the same starting point (from the 'ground up' in this example), then the recursive case will probably require less debugging effort overall than the non-recursive case. On the other hand, if this restriction is not considered, then the non-recursive case may frequently require less investment in the debugging effort thanks to the fact it may make use of previously-debugged, known correct sub-specifications for its com-
ponent transformations, which is not as relevant in the recursive case.

2.2.5 The Empirical Debugging Cycle: An Iterative Convergence Mechanism to a Correct Transformation

The empirical debugging process actually corresponds to a repetitively-executed cycle of activities, which have as their goal the incremental modification of the implemented input/output transformation in order to bring it into conformance with the 'correct' transformation envisioned by the programmer/user for a particular application. I term this cycle of activities the empirical debugging cycle (or, just debugging cycle).

For purposes of this discussion, I will view the abstract input/output transformation, as it exists on a real machine, as being completely characterized at any given point in its execution by an instantaneous description consisting of its execution state and its so-called abstract data state. The former is the usual definition relating to the 'running'/‘not-running’ status of the system process corresponding to the implemented transformation. The latter, however, is a little unusual in that it is defined to be the contents of all mutable storage elements associated with the transformation, including program instruction text, as well as data areas. The reason for this special definition of abstract data state is that, in the debugging world, text elements can be viewed on the same level as data elements (including registers), in the sense that the contents of both are equally accessible and mutable by external means.⁹ Therefore, with this characterization, the abstract concept of performing modifications to the input/output transformation translates to the concrete concept of modification of the data state of the corresponding program.

Now, the individual activities involved in the debugging cycle can be grouped into a schedule comprising three generalized categories of activities, as follows (also, see [AD71]):

⁹ For example, consider the general peek and poke operations, which operate not only on a program's data space but also its text space
• observe the actual execution state and abstract data state (including outputs, as appropriate) of the program, and compare them with the corresponding expected values, as envisioned by the programmer for the ‘correct’ transformation, in order to produce a kind of abstract ‘error difference’ between the two cases

• modify the abstract data state of the program, based on the abstract error difference determined in the previous step, in such a manner as to correct or diminish the error difference in the next iteration of the debugging cycle (this is the ‘negative feedback’ step)

• modify the execution state of the program, in order to facilitate the next iteration of the debugging cycle (note this could involve stopping as well as restarting execution of the program, depending on the particular situation)

Of course, it is important to recognize that for any particular situation or even specific iteration of the cycle, one or more of these steps may be skipped. In particular, this may be true for the last step involving the modification of the execution state of the program.

In one sense, a fundamental goal of programming is to utilize the automated capabilities of the machine in the task of completing an initially incomplete (in the sense of ‘erroneous’ vs. ‘correct’) specification of an input/output transformation provided by the programmer. Consequently, the goal of the iterative debugging cycle is eventually to cause the current version of the input/output transformation (whose initial ‘value’ was the result of the erroneous specification provided by the programmer in the first place) to converge to the correct version. That is, the current transformation effectively ought to become more and more ‘correct’, in the sense that the measured abstract ‘error difference’ between the observed and expected values of the execution and abstract data states of the program becomes progressively smaller in magnitude (according to some reasonable metric) with each successive pass through the cycle. Note that, as with any mathematical convergence process, the rate of convergence depends on the specific approaches used in implementing each of these steps. Assuming that the debugging agent is a human, the approaches employed are, of course, the product of the human thought process, and the rate of convergence greatly benefits from special ‘oracle’ insights, or the ‘Aha!’ phenomenon. On the other hand, insofar as automatic (mechanical) debugging agents are concerned, it is in-
teresting to note that some of the elements of the debugging cycle are often incorporated into modern compilers and interpreters in one form or another, at least for the detection and correction of syntax errors, and sometimes even for relatively simple semantic errors. Of course, the compiler does not actually execute the program in order to debug it, but indeed does use combinations of rules and heuristics, and possibly (in the case of low-level semantic errors) even real-time interaction with a human in a process akin to simulation of (portions of) the program.

As the number of iterations increases and the current transformation becomes a better approximation to the correct transformation, the 'observability' of the remaining errors will in generally decline. This effect results in a decrease in the rate of convergence of the empirical debugging process, in terms of the effective number of errors corrected per iteration of the debugging cycle (which is a number possibly much less than 1). In other words, a point can be reached in the empirical debugging process for 'large' transformations at which the remaining errors in the current transformation effectively become unobservable, at least in the sense that the errors do not readily manifest their presence in the outputs resulting from the particular set of test inputs being used for the debugging process. Consequently, the number of input test cases required in order to observe and correct a remaining error will increase as the debugging process proceeds. In addition, the rate of convergence can decrease as a result of the increasing difficulty of localizing and correcting within the data state of the current transformation, those remaining errors whose effects are in fact observable in terms of their effects on the outputs.

Therefore, the fewer and more subtle the remaining differences between the current and correct versions, the more explicitly one needs to know detailed information both in regards to the measured as well as the expected (predicted) internal data states, in order to be able to observe experimentally a specific point of difference (in accordance with the subtlety principle, as discussed in a previous section). In fact, the derivation of the predicted internal data state alone can become quite a difficult task at some level.

---

10 This is in accordance with the decreasing 'observability' of the remaining errors, and the correspondingly decreasing rate of convergence for the debugging cycle.
One valuable technique that is useful in this sort of situation is simple reconnoitering of the internal data state through incremental (‘stepped’) execution, as applied both in the forward and reverse directions, whereby one starts from a known intermediate (or even the final) data state and works in the respective direction towards the point at which it is suspected that a difference will be found. Once a specific point of difference is observed, then it is usually a straightforward matter to map to the actual corresponding error source in the specification. If the specification is structured according to some hierarchy of lower-level sub-specifications, then this process usually will be conducted on a top-down basis, proceeding from the highest level of abstraction to the lowest, in order to systematically determine ever-narrowing boundaries within which the error is localized. Furthermore, as one descends from higher to lower levels of abstraction in this manner, the experimental granularity of the observed data state is decreasing, and the aforementioned ‘stepped’ execution correspondingly should be carried out in accordance with the current level of abstraction being utilized. Thus, at the highest level of abstraction, a ‘step’ would involve abstract execution, as objects, of the high-level sub-specifications constituting the overall specification, and each such step will effect a relatively large incremental change in the data state. In contrast to this, at the lowest possible level of abstraction, a ‘step’ would involve execution of the low-level machine instructions equivalent to the original specification, and each such step will effect only a relatively very small incremental change in the data state. Also, at any given level of abstraction, (re)execution of an arbitrary step can be implemented by means of a combination of special logging techniques and dependence analysis techniques relating the data dependences of the pre- and post-data states on either side of each step boundary (for an example of this approach, as formulated at the level of abstraction corresponding to the programming language constructs, see [MC88a] and [Sto88]).

Note that most debugging tools are geared for a limited range of levels of abstraction, usually no higher than that level inherent in the basic constructs of some specific programming language (i.e., ‘source-level’ debuggers) However, debugging a specification at the very highest levels of semantic abstraction (i.e., the algorithmic level) is almost exclusively carried out within the human head. Therefore, the ideal debugging tool would be one capable of abstracting a lower-level specification (e.g., at the conventional programming language level) into a semantically equivalent, higher-level specification that could be more easily verified as ‘correct’ by the programmer. Note that, more than likely, such tools would have to be based upon prin-
ciples from the area of artificial intelligence, as further discussed below.\textsuperscript{11}

At some point, the programmer must make a \textit{judgement} that the transformation is \textit{sufficiently 'correct'}, based on the observed behavior of the transformation in the debugging cycle, and begin to allow users to operate the 'debugged' version of the program in \textit{production} mode, corresponding to the generation of outputs from inputs in the application for which the program was designed. In accordance with the above comments, any such program is an \textit{approximation}, to some degree, to the 'correct' version. Now, the debugging cycle does not really terminate after the 'debugged' program is issued for use in its intended application. Instead, the mechanics of its execution have been simply altered, such that the responsibility for carrying out that part of the first step of the debugging cycle involving observation of actual \textit{vs.} expected outputs has been shifted from the \textit{programmer} to the \textit{users}. Thus, the distinction between 'development' and 'production' modes for a program simply involves the alteration of the makeup of the debugging agent to include multiple observers in the latter case. However, the debugging cycle itself remains fundamentally unchanged (although in some sense it has been 'expanded'), and iterations continue under production mode, wherein the first step of the cycle is carried out in the new form involving feedback of 'bug reports' from users to the programmer. Subsequently, the programmer carries out the debugging cycle to resolve the user-observed 'bugs', and reissues an updated version of the program, which hopefully is an even better approximation to the completely correct version, for use in production mode.

Now, one might well imagine that the great challenge is to construct a mechanical debugging agent to carry out the debugging cycle (in 'development' mode), given as inputs the initial 'buggy' program, plus some sort of alternative specification(s)\textsuperscript{12} for the 'correct' input/output transformation envisioned by the user. The latter could

\textsuperscript{11} Of course, one might argue that a better alternative to developing such a debugger would be to develop and use an even more abstract, high-level programming language. In any case, this programming language likely would be very specialized, with constructs specifically adapted for a targeted, narrow class of applications. However, an argument in favor of the abstracting debugger would be the 'inertia' argument — that is, it is justified by the large investment already made in programs developed using existing programming languages.

\textsuperscript{12} For example, \textit{loop invariants}. 
be drawn from a spectrum of possibilities. At the low end, it could take the form of a set of expected (correct) input/output pairs, or examples, which is the basic idea behind empirical debugging as discussed herein (also reference the BIP [Bar76] and PDS8 [Sha83] debugging systems). At a somewhat higher level, the alternative specification could consist of a set of analogies, plus the expected deviations therefrom for the current transformation to be implemented. For example, a known correct specification for a transformation that is analogous in behavior (viz. inputs/outputs) to the target transformation could be supplied, along with a precise catalog of behavior deviations (as practical) expected in the new transformation, possibly expressed in the form of rules or even specific examples. At the highest level, the alternative specification would take the form of restatement(s) of the program in terms of high-level abstractions (for example, reference the HUDSY [Luk80] debugging system).

However, note that the effects of errors in any form of the alternative specification itself would naturally introduce even greater uncertainty in the correctness of the ‘final’ product. So again, it must be reemphasized that the debugging cycle is both empirical as well as stochastic, at least in the sense that the final debugged program can only be assumed ‘correct’ within a certain probabilistic confidence interval.

Observe that such a fully mechanical debugging agent is in fact very close to being a fully automatic program development system, which will ideally produce a verifiably correct input/output transformation given a very high-level form of abstract specification (e.g., an English description). Also, note the similarity between the manner in which a human would interact with the automatic debugger (viz. provision of alternative specifications) and the manner in which a teacher relates to a pupil attempting to devise a correct solution to some problem. Undoubtedly, therefore, any successful mechanical debugging agent will have to incorporate elements from the area of artificial intelligence (AI) in its design, in order to infuse it with sufficient capability to deal with more than simply the most trivial of programs (this theme is continued informally in section 2.5).

In connection with this, it is now an accepted dogma (e.g., reference remarks by Fritzen [Fri84]) that any programming environment system must contain, as its central tool, at least a semi-automatic programming/debugging system, which involves a hybrid of human interaction and automated machine actions in the program
development/debugging cycle \textit{(i.e.,} a hybridized man-machine debugging agent, with heaviest emphasis, naturally, on the former\textit{)}. Of course, many semi-automatic programming systems have been developed in recent years, including most notably the Cornell Program Synthesizer [TR81]. Many \textit{(if not most)} of these systems are specifically concerned with the prevention or elimination of syntax \textit{(versus} semantic) errors. However, some recent systems have begun to incorporate elements from AI, most notably combinations of heuristics and rule-based analysis methods, in order to bridge the gap and \textit{(at least interactively)} provide the human programmer/debugger with insights into possible problems with the semantics of a specification, as well. One example of such a system is \textbf{Talus} (see [Mur88]), an 'intelligent' tutoring system for student programmers, that provides mechanical debugging support through use of the program verification approach. In particular, this system utilizes a version of the Boyer-Moore theorem prover (see [BM79]) to attempt to verify that LISP-like code developed by a student as a solution for a given problem in fact does correspond to \textit{(i.e.,} is semantically equivalent to) some known-correct \textbf{reference solution} to that problem. Other such semi-automatic, 'intelligent' debugging systems (as summarized by Murray in [Mur88], section 2.3) include:

\begin{itemize}
  \item \textbf{PDS6} ([Sha83]) — Provides interactive debugging of PROLOG-like programs, primarily based on: (1) the detection and correction of bugs belonging to so-called \textbf{bug equivalence classes}, which are constructed according to examples of typical types of bugs furnished by the user; (2) monitoring of the stack size in order to detect infinite loops; and, (3) queries directed at the user \textit{viz.} the correctness of procedure traces, in order to verify that the \textit{observed} program trace corresponds to the \textit{expected} program trace.

  \item \textbf{PROUST} ([Joh85]) — Provides static debugging analysis of student programs based on heuristic detection of such a program's conformity or nonconformity to a set of \textit{plans} stored in a 'plan library'. This approach is termed by Murray [Mur88] the 'heuristic plan recognition' approach. Note that these 'plans', or intended goals for the program, simply constitute an alternative, high-level form of specification for the program and/or its constituent subprograms, as alluded to above.

  \item \textbf{Pudy} ([Luk80]) — In addition to utilizing some of the same techniques as employed by \textbf{PDS6} and \textbf{PROUST}, \textbf{Pudy} provides static analysis of a
program's correctness based upon a determination of whether the program is consistent with a set of alternative, high-level specifications that take the form of predicate calculus assertions about the program, including loop invariants and assertions about the input/output computational domain.

2.3 Review of the Basic Elements of Debugging

The debugging process is typically highly empirical, relying largely on experimental observations of actual versus expected outputs for a given set of inputs. In accordance with the aforementioned empirical debugging cycle, an iterative convergence process is carried out, involving modifications made to the internal 'data state' of an input/output transformation in order to implement corrective compensation for these deviations of the 'actual' from the 'expected'. Thus, there must be tools available in any debugging system to effect these modifications, as well as modifications to the 'execution state' of the transformation, and to facilitate observation of the results of such modifications. These tools may be implemented at any conceivable level of abstraction, but ideally should be specifically appropriate to the level of semantic complexity of the original specification for the transformation. However, most modern debugging tools are oriented towards performing the debugging task specifically at the level of abstraction inherent in the programming language constructs used to formulate the original specification.

Thus, many debugging tools are very low-level, in that they are primarily designed for debugging of 'bits' and 'bytes'; that is, observation and modification of the physical data resident in the machine's memory and register hardware. An example of this type of debugging system is the UNIX adb general-purpose debugger. On the other hand, more 'advanced' debugging tools are designed to uncouple the programmer from the task of relating a high-level, abstract specification to its corresponding low-level, architecturally-dependent representation on the substrate machine. Most such advanced debugging tools are of the so-called 'source-level' variety, which allow the programmer to perform the empirical debugging task in the context of the the same programming language constructs used to create the original specification (program source text). An example of this type of system is the UNIX dbx source-level debugger. However, as touched upon above, the next generation of
advanced debugging tools will make the transition from being oriented towards the relatively low-order semantics of conventional programming language constructs, to being oriented towards revealing the higher-order semantics embodied in a lower-level specification, and likely will incorporate elements from AI in order to facilitate this truly abstract debugging.

2.3.1 The Relationship Between Code Translation and Debugging; Translation Boundaries

From the standpoint of a typical programmer, a fundamental measure of the utility of any debugging tool is the relative degree of cross-referencing (or mapping) required on his part, between the abstract data states (see definition and discussion in section 2.2.5 above) of his input/output transformation as set in the context of the original, higher-level specification versus that of the corresponding lower-level specification generated by some translation process (e.g., some phase of compilation). The debugging tool should ideally perform this mapping completely and automatically. Therefore, in this sense, debugging can generally be considered as the inverse operation of the usual translation process that transforms a sequence of higher-level abstractions constituting the original specification (e.g., program source text) to some sequence of lower-level abstractions constituting the semantically equivalent but 'simplified' (viz. semantic complexity) specification (e.g., the architecturally-dependent code executed on the substrate machine). In fact, this concept is reflected in the appropriate and familiar terminology, 'dis-assemble' (viz. low-level machine code debugging systems), or 'de-compile' (viz. source-level debugging systems), as used to describe the process of mapping the lower level to the higher level of abstraction. The greater the extent of the mapping task performed automatically and internally by the debugging tool, the greater the effective utility of the tool from the standpoint of the programmer. Thus, the idealized debugging tool should be able to provide the programmer with an invariant view of the temporally evolving abstract data state of his input/output transformation across the translation boundary separating layers (or, abstraction environments) characterized by higher and lower levels of abstraction, and that correspondingly contain equivalent high- and low-level specifications for the transformation. These concepts are illustrated in Figure 2.2a.

Here, 'invariant view' is used in the sense that, for a given abstraction environment, the associated representation of the abstract data state of the transformation
always remains at the same level of abstraction as that used in formulating the specification contained in that environment, regardless of any additions, deletions, or changes with respect to other environments or translation boundaries. However, this does not always mean that the high-level representation of the abstract data state on one side of a translation boundary continuously and automatically 'tracks' the lower-level representation on the other side, nor vice-versa. Rather, the update of abstract
data state information from one environment to the other, via either the 'translate/modify' or 'de-translate/inspect' function, as appropriate, can be effected according to several different strategies, depending on the system design goals. These include:

- synchronous auto-update based on a fixed time period, particularly with respect to those elements in the abstract data state that have changed during that period
- asynchronous auto-update, triggered whenever any element (or a specific element) changes in the abstract data state
- manual update on demand, targeting either randomly specified elements of the abstract data state, or all elements that have changed, or simply all elements, etc.

For example, in the case of a 'translate/modify' function that corresponds, say, to an incremental compilation step, then one of the latter two strategies likely would be employed.\(^{13}\)

As mentioned, each abstraction environment on either side of a translation boundary is characterized by a certain level of abstraction. In concrete terms, this level of abstraction is embodied in some sort of corresponding 'language' that is used to formulate the specification contained within that environment. In other words, each environment has associated with it some 'language' whose constructs form the basis for expressing — at the appropriate level — abstractions within that environment. Likewise, in a given environment, the representation of the abstract data state for an active input/output transformation is expressed and understood in terms of the language constructs associated with that environment.

Similarly, at a given level of abstraction, the initial form of the representation of the abstract data state directly corresponds to the form of the specification at that level. In a sense, therefore, a specification is simply an initial value for the abstract

---

\(^{13}\) Perhaps more than one update strategy could be made available for each such function. The selection of which strategy to use for a given function on a given translation boundary then could be performed dynamically by the system or the programmer, in accordance with what is deemed necessary and/or expedient in a given situation.
data state representation contained in an environment, which subsequently will evolve — *via* some update strategy, as discussed above — in connection with the execution of the underlying input/output transformation. For example, in terms of abstract data states, 'compilation' is simply that phase of translation, as effected *via* the 'translate/modify' function, that initializes the corresponding lower-level abstract data state, given the already-initialized, high-level abstract data state. Subsequent evolution of the lower level abstract data state can be reflected into a corresponding evolution of the higher level abstract data state *via* operation of the 'de-translate/inspect' function.

Note that the basic 'translate/debug' relationship shown in Figure 2.2a could be repeated to form one of many such stages, involving multiple translation boundaries and levels of abstraction, all coupled in a linear chain fashion with the highest level of abstraction on the left, say, and the lowest on the right. This is illustrated in Figure 2.2(b).

The programmer's interaction with such an extended chain would ideally still be at the far left, as well, corresponding to the highest level of abstraction. However, depending on the situation, the programmer nevertheless sometimes will interact with a different stage in the chain than the leftmost. For example, the second stage could represent a 'source program development environment', through which the programmer normally would carry out entry of and changes to program source text in terms of some relatively high-level programming language. The leftmost stage then would represent some higher-order 'applications development environment', containing an equivalent specification formulated in terms of some some set of abstracted *function objects* and *data objects* that are specially tailored for the given application or class of applications.¹⁴

---

¹⁴ For this example, note that the ideal situation would be that in which the set of all such abstract, application-oriented objects would be both correct and (sufficiently) complete in their definition, so the programmer would be able to carry out the debugging process for the given application entirely at this most highly abstract level, interacting with the system only in terms of instances of these objects. Realistically, however, such abstract objects, which were defined at the (lower) source text level, probably will need to be modified or deleted, and new types of objects created. Thus, the translation of these changes in the abstract specification into equivalent changes in the concrete source-level specification must be done manually by the programmer. That is, the 'translate/modify' function connecting the two environments cannot be purely mechanical, and the programmer must
In accordance with this example, one might have a nine-stage abstraction scheme of environments for specifying or expressing the abstract data state of a transformation in a distributed programming situation involving optimizations, where the abstract data state of the transformation in the $n^{th}$ stage is contained in an environment $E_n$, characterized by the 'languages' based on the following constructs:

E1: Applications Development Environment :: Abstract Function/Data Objects

E2: Source Program Development Environment :: Source Language Code

E3: Intermediate Environment #1 :: Intermediate Language Code (_ILC)

E4: Intermediate Environment #2 :: Optimized _ILC

E5: Intermediate Environment #3 :: Interprocessor Transport Language Code (_ITLC) — machine # 1

E6: Intermediate Environment #4 :: Interprocessor _ITLC — machine # 2

E7: Intermediate Environment #5 :: Optimized _ITLC (i.e., machine-specific optimizations)

E8: Machine Language Environment :: Native Machine Language/Assembly Code

E9: Run-Time Environment :: Machine Objects

Note that the translation boundary and associated 'translate/modify' and 'de-translate/inspect' functions assume respectively appropriate meanings between each of these stages. In particular, the translation boundary between stages 5 and 6 is the inter-machine communications network.\(^{15}\)

---

\(^{15}\) For this example, note that the representation of the abstract data state on either side of this boundary, in terms of abstractions expressed in Interprocessor Transport Language code (ITLC), will be the same on both machines, even if the two machine architectures differ (also refer to the discussion in section 2.4.2 concerning inter-architecture data compatibility conflicts).
Now, the translate/debug relationship is especially relevant in the context of parallel/distributed programs running on machines with different architectures in a distributed network (which is, in fact, a fundamental assumption of this thesis). In this situation, there is at least one translation boundary (and likely many, in similarity with the preceding multi-stage example) separating the program development environment, in which programming language constructs constituting the program source text are the basic elements of the higher-order level of abstraction, and the remote run-time environment, in which architecturally-dependent machine instructions and machine objects — e.g., memory addresses, registers, etc. — are the basic elements of the lower-order level of abstraction. Especially as the result of the fact that the machine architectures on opposite sides of the translation boundary can be different, the mapping process involved in the translate/debug relationship is potentially both complicated and inefficient.\(^\text{16}\)

Furthermore, consideration of optimized code complicates the picture by introducing one or more intermediate layers between the existing program development and run-time environments, each of which corresponds to some intermediate level of abstraction appropriate for facilitation of the optimization process. Each such layer introduced will of course be defined by translation boundaries that consequently require the existence of appropriate mappings in order to carry out the debugging process at the highest possible level of abstraction. Again, the complexity and potential inefficiency involved in performing these additional mappings across the new translation boundaries is at some point made even more intense in view of the likely possibility that the program development and run-time environments are present on separate machines potentially having different architectures.\(^\text{17}\)

---

\(^{16}\) This issue is in fact a large part of the motivation behind the work described in subsequent chapters of this thesis, involving the development of a new facility for remote execution and debugging in such distributed, inhomogeneous computing environments.

\(^{17}\) This issue also is relevant in the discussion in Chapter 5 that pertains to future research in the area of debuggers capable of dealing with optimized, parallel/distributed code.
2.3.2 Basic Elements of Execution Control in Debugging

For debugging purposes, the basic elements of execution control consist of primitive operations that effect the run-time start, stop (temporary suspension), single-step, continuation, and termination of a program. These primitives may be targeted for operation at one or more abstraction levels — for example, at the very low-order machine-code level, or (better) at the level of the programming language constructs used in the original specification (ref. 'source level' debugging systems). For example, the single-step and continuation operations take as an operand a 'location' at which to effect the corresponding action; however, just what constitutes a 'location' is dependent on the level of abstraction for which the debugging system is implemented. Thus, for machine-code level debuggers, a 'location' is an explicit memory address in the translated (compiled) machine code; for source-level debuggers, a 'location' is usually a statement boundary in the original (untranslated) source text. A source-level debugging system must therefore be able to internally and automatically map between addresses of machine instructions and source statement numbers. Such an inverse mapping is the reverse-direction 'bridge' across the translation boundary between executable machine code and abstract program source code. The forward-direction bridge is the inverse of this mapping relation; namely, the mapping of the sequence of source statements into a corresponding sequence of machine instructions as the result of compilation (translation). This complementary mappings relationship was described in the previous section (ref. the translate/debug relationship).

Note that the stop (suspend execution) operation takes two forms. The first is asynchronous in the sense that it is issued by the programmer at some random point during the execution of a program. The second, the ubiquitous breakpoint, is synchronous in the sense that the programmer embeds a preset stop operation directly into the code at some precise location, in order to 'trap' the program at that point in its execution (for example, to permit observation of the experimental abstract data state of the program, and subsequent comparison to the expected data state as part of the

---

18 Note that, in the discussion to follow, the term 'program' will be used in its usual, though arguably semi-ambiguous context, replacing the more intuitively precise yet syntactically more cumbersome notion of a 'concrete implementation of an input/output transformation'.
previously-discussed empirical debugging cycle). The stop action will be effected when the machine attempts to execute the instruction at that location, and the debugger then has the responsibility (as with any and all program execution state changes) of reporting the event back to the programmer. As with the continuation operation, the semantics of the 'location' specified as the operand of the breakpoint operation is dependent on the level of abstraction for which the debugging system is implemented — for source-level debuggers, a breakpoint typically may be set (and reported as such by the debugger when the breakpoint event occurs) at the beginning, middle, or end of any executable statement in the source text. Breakpoints are usually implemented using some form of special trap or illegal machine instruction that is substituted for the program code normally present at the specified location. However, the semantics of an installed breakpoint in high-level debugging systems may be such that any explicit attempts by the programmer to inspect the instruction contents at the location of the breakpoint will return the original program code that was present there.

In addition to this 'normal' form of the breakpoint, some advanced debugging systems permit the use of so-called conditional breakpoints. This type of breakpoint is designed to be triggered upon some special combination of events, which is specified by the programmer using the appropriate predicates.

2.3.3 Basic Elements of Data State Observation and Modification

Recall that, in for the purposes of this thesis, the abstract data state of a program is defined as all mutable storage elements associated with the program, including program instruction text, as well as data areas (viz. both memory and registers). The basic elements of data state observation and modification in most debugging systems consist of primitive operations for inspecting and mutating the contents of a particular location in a program (i.e., viz. instructions, data, or registers); namely, generalized peek/trace and poke operations, respectively. Note the difference between a peek and a trace operation is that peek is an explicit, user-invoked operation to observe the contents of a location, whereas trace is like an automatic peek embedded in the program, which reports the contents of a location upon being implicitly triggered by the detection of a software-invoked access attempt to the location during the normal course of execution of a program. (This automatic triggering can be most efficiently implemented via special support from the operating system and/or
hardware; however, it can be implemented via explicit stop/peek/continue sequences targeted at the specified locations, though much less efficiently.) Once again, the semantics of the 'location' specified as the operand of these operations is dependent on the level of abstraction for which the debugging system is implemented. For example, a high-level debugger would ideally permit specification of a data 'location' using the direct abstraction of the actual symbolic name of the associated data structure. Similarly, such a debugger might permit symbolic abstraction of the program data stack, in particular stack call frames, in an architecturally-independent manner. In addition, the semantics of the 'contents' of a program 'location' is similarly dependent upon the level of abstraction for which the debugging system is implemented. For example, a debugger might permit peek and poke operations on abstract, typed data objects (e.g., a 'local' floating-point array variable in a subroutine, located on the dynamic program data stack), independent of the system-dependent, physical representation of the data on the machine.\(^\text{19}\)

Once again, note that the successful implementation of these debugging features is really centered on the issue of mapping. In other words, cross-referencing between representations of the abstract data state of a program that correspond to different levels of abstraction — a 'high' level appropriate for the programmer, and a 'low' level appropriate for the machine. Therefore, a primary distinguishing feature between high- and low-level debuggers is the facility within the former to extensively carry out automatic mappings between physical, machine-oriented objects of very low innate semantic complexity and abstract, programmer-oriented objects with semantic complexity of at least the same order as the programming language constructs (statements/instructions and data structures) constituting the original program source specification.

\(^{19}\) \textit{i.e.}, the representation \textit{viz.} the number of bytes of precision, byte order, sign representation, etc.
2.4 Special Issues in Parallel/Distributed Debugging

2.4.1 Synchronization Conflicts and the Probe Effect

A common and consternating source of errors that is that of a unique to parallel code is timing or *synchronization* conflict. For shared memory multiprocessor (SMMF) systems, this is specifically the case in regards to parallel code dealing with shared data structures. For a distributed, message-passing based environment, shared memory is not the issue; rather, it is one of concurrency and process coordination.

The sources for errors in the timing relationships between parallel code segments are often very subtle and difficult to localize. In fact, synchronization effects are frequently very sensitive to external conditions, such as I/O availability and timing. As another example, on an asynchronous (or semi-synchronous) multiprocessor or multicomputer (distributed) system, random variations in the execution scheduling of the individual component processes corresponding to the parallel code segments can occur, an effect largely attributable to decisions made by the operating system(s), as well as to nondeterminism in system hardware/software (e.g., clock frequency and phase). As a consequence, the parallel program may behave in a nondeterministic manner from one execution instance to another, displaying 'spurious' or 'intermittent' errors in the produced results (this is particularly a problem in regards to the presence of *race conditions* connected with the near-simultaneous modification of the contents of shared or common data structures by different parallel code segments). Furthermore, artificial interference from a debugging system can affect the timing relationships between the parallel code segments, possibly disguising, altering, or even creating synchronization conflicts — this is known as the *probe effect* (reference Gait [Gai85] and [Gai86]).

---

20 Note that this problem is especially relevant in regards to breakpoints, since a breakpoint that stops a single process can play havoc with the overall timing picture. About the only thing that can be done to remedy the situation is to halt *all* the processes at the same time — but unless this 'same time' is just that, then the resulting global state can be inconsistent (in the sense of deterministically repeatable). However, it is nevertheless possible to effect a global halt that results in a consistent global state over all the processes, even in the worst-case scenario involving loosely-coupled, asynchronously-running, communicating processes, as a special debugging-support feature that is provided by a suit-
Therefore, synchronization conflicts are a major factor complicating the debugging task for parallel code. This is especially true with respect to the nondeterminism displayed in the occurrence of observable effects of the conflicts, as modulated by external factors such as scheduling (viz., operating system and system hardware/firmware) and interference from a probe (debugger). Such nondeterminism often can effectively cripple the empirical debugging process, which relies on the ability to carry out iterations of a debugging cycle involving experimental observations on the abstract data state of the program (for comparison to expected values).

Nevertheless, there are several general approaches that can be used to debug synchronization conflicts. One example is an off-line symbolic approach based on dependence analysis, which is especially applicable in the case that the original source text is written in a sequential form that is subsequently parallelized, either manually or automatically. In this approach, the (symbolic) debugging system attempts to determine whether potentially parallelizable text segments (e.g., loop iterations) can in fact be executed successfully in parallel without the introduction of synchronization conflicts. Also, since this determination is performed at the symbolic level on the basis of dependence analysis, then it may be combined with certain optimization techniques for parallel code that also are based on dependence analysis. Active research in this area is underway at Rice University (reference [ACK87], [AK87], and [AK82]; also, see [Hen87]).

On the other hand, there are several run-time based, empirically oriented approaches for debugging synchronization conflicts in parallel/distributed programs. One such approach involves the direct determination — via a combination of static analysis and experimental observation — of the actual ordering of concurrent events in a parallel program. This information then allows the programmer to determine the causal relationships between the concurrent events, and thereby to examine whether race conditions may be occurring between cooperating processes. In particular, reference [MC88b].

21 Note that an off-line symbolic approach based on control flow analysis would be tantamount to simulating the execution of the parallel code, which in most cases would likely be an inefficient and costly approach, and in many cases is in fact useless because the nature of any potential timing conflicts would almost certainly be altered within the simulation!
lar, Miller and Choi (see [MC88a]) have implemented this approach based on the evaluation of the so-called 'Parallel Dynamic Dependence Graph' (PDDG) for the set of cooperating, parallel/distributed processes. Note, however, that this evaluation depends not only on some static symbolic analysis of the original code, but also on special logging information, generated at run-time by specialized debugging code fragments inserted into the original program text by the (modified) compiler.\textsuperscript{22}

Another empirical, run-time approach involves manually permuting and enforcing synchronization points in the program — for example, experimental insertion/deletion/relocation of explicit \texttt{synch} or rendezvous points in order to force the manifestation of an expected/unexpected behavior in the program. Thus, this approach entails physical modifications to the program code. Furthermore, these modifications are performed at the level of abstraction corresponding to that of the \texttt{synch} construct.

Of course, both of these run-time approaches seem susceptible to the pitfalls of the probe effect\textsuperscript{23}, since they both involve physical modifications to the program code.

\textsuperscript{22} These specialized code fragments operate in conjunction with the operating system to record/log information pertaining to causal relationships between synchronization events, such as the sending or receiving of a message by a process in the case of a loosely-coupled, distributed system.

\textsuperscript{23} Note that in their paper [MC88a], Miller and Choi tend to gloss over any negative impact of their approach with regards to the probe effect. Instead, they emphasize the relatively small magnitude of degradation in the execution performance of parallel programs debugged using their technique (\textit{e.g.}, approximately a 10 to 15 percent increase in execution time in most cases). Also, their approach, by its very nature, involves the storing and manipulation of large amounts of additional logging information — however, the positive aspect emphasized here is that, for a given parallel program, the relative amount of such logging information produced by their technique is greatly reduced in comparison to that produced for the naive debugging scenario, in which virtually everything about the parallel program is traced/logged.
2.4.2 Data Compatibility Conflicts in a Distributed, Inhomogeneous Computing Environment

In a distributed computing environment, communications between computing elements constituting the environment is carried out by means of discrete message packets. If the computing elements are not all alike from the standpoint of their internal, physical representation of the various primitive data types (i.e., due to fundamental differences in architecture), then the distributed environment is termed inhomogeneous. Consequently, in such a situation, (many of) the typed data items communicated in message packets passed between the elements will be incompatible in terms of their physical representation, giving rise to a data portability problem. For example, the architectures involved may differ in terms of byte order, word (integer) length, and/or the format of some of the simple primitive data types (e.g., integer sign representation, floating point mantissa and/or exponent representation, etc.). Note that, as used here, 'simple primitive data types' refers to those primitive data types that are usually assumed to map directly onto hardware — namely, byte/char, integer, floating point, pointer/address, and word (unsigned integer). ‘Compound’ or multiple-precision primitive data types (i.e., those primitive data types whose implementation is on some systems software- or firmware-dependent, such as short/long integer or double precision floating point) can also present an incompatibility problem due to differing precisions/lengths, as well.

There are (at least) two levels of data portability problems created by typed data incompatibilities. The first involves the obvious low-level issue of successful transportation of the incompatible data from one computing element to another. This problem is superficially solvable in a straightforward manner — namely, simply transform the data as appropriate (e.g., *viz.* byte order). However, the real solution is not quite as simple as it appears on the surface, being complicated largely by the fact that the *precisions* for certain primitive data types may differ for different computing elements in the distributed environment. Thus, depending on the data type, data transportation might involve either partial loss or even corruption of the information content. As an example, the internal precision for a particular numerical data type on the receiving end may be less than that on the sending end, so that the transported data item simply won’t ‘fit’ on the target element without rounding or truncation (corresponding to a loss of information). Similarly, a transported pointer value (i.e., memory address) can be completely corrupted by the truncation required to make it
'fit' into a smaller length representation for pointers on the receiving end.

Furthermore, simplistic transformation of typed data items between different computing elements (machines) does not always preserve the intended semantics of the information actually stored in the data items (even when the two precisions or lengths are the same), in particular when the intended semantics is somehow related to the physical structure of the specific representation for the data item on a particular architecture. That is, a naive solution that simply 'transforms the data' can ignore the significance or meaning of the implicit information content stored within the data item. In particular, this situation can be brought about whenever a data item having a particular physical data type is considered to also have an alternate, logical data type (i.e., as with overlays of data items having different types). For example, consider the data structure overlays and code fragments written for two separate machines (in the C language), as shown in Figure 2.3.

Assume that the size of integers on both machines is the same, and that the fetch routine on machine 'B' properly blocks until the integer value for t.n is received from the send routine on machine 'A'. Now, the number "65" is the decimal ASCII equivalent for the character "A", and is loaded into the arithmetically least significant byte (LSB) of the integer t.n on machine 'A'. Also, it is assumed by the programmer that machine 'A' has a low-to-high physical byte order, so that the printf() statement on line 6 prints "65, A". If both machines have the same (low-to-high) byte order, then the printf() statement on line 13 for machine 'B' will also print out the same thing, namely "65, A". However, if machine 'B' has a high-to-low byte order, then the result printed out by the printf() statement in line 13 will (incorrectly) be "65, <nil>". This reflects an incorrect expectation for the character value "A" to be found stored in the character array position t.c[0] (as it was on machine 'A'), even though the (overlaid) integer value in t.n is in fact correct ("65") thanks to the automatic byte reversal implemented for integers as part of the send / fetch operations! In other words, in this situation, t.c[0] overlays the arithmetically least significant byte (LSB) of t.n on machine 'A', but on machine 'B' it overlays the arithmetically most significant byte (MSB). The point is, the data transformation that in this situation correctly preserves integer values (i.e., byte reversal) cannot anticipate the semantics of the overlaid (character) data stored within the integer. In fact, for this example, it is impossible for any low-level data transformation to simultaneously preserve both the integer value and the character value, without changing the semantics of the C language (viz. union) itself!!
machine A:

```c
1 union {
2     int n;
3     char c[sizeof(int)];
4 } t;
...
5 t.n = 65;    /* stores integer equiv. of char. "A" in LSB */
6 printf("%d, %c", t.n, t.c[0]);
7 send_integer_value_to_machine_B(t.n);
...
```

machine B:

```c
8 union {
9     int n;
10    char c[sizeof(int)];
11 } t;
...
12 fetch_integer_value_from_machine_A(&t.n);
13 printf("%d, %c", t.n, t.c[0]);
...
```

Figure 2.3: Example Showing Loss of Information Content

Now, the problem of manipulating incompatible data items is especially relevant from the standpoint of remote debugging. At the heart of the empirical debugging process is the fundamental need on the part of the programmer to be able to control the execution state, and inspect and/or modify the internal data state of a program.
If the programmer is situated at one machine, and the remote program being debugged is situated on another, architecturally incompatible machine, then clearly the programmer needs to have some sort of consistent view of the various remote data items, even though their physical representations on the two machines might be incompatible. In other words, in this situation, it makes no sense to require the programmer to understand all the architectural details pertaining to both machines, such that he himself is required to manually fetch and interpret the remote data at the level of the architecturally least common denominator between the two machines — namely, the lowest possible level of abstraction, bytes. Rather, automatic transformations ought to be implemented that abstract away the architectural differences between the two machines. The natural candidates for these abstractions are at the first level of abstraction above the byte level — namely, the primitive data types whose representations conflict on the two machines. Thus, the data transport mechanism of the debugging system should automatically (and ‘invisibly’) perform the necessary transformations on the data to accomplish this task, in a manner easily adaptable to many types of architectures. One possible way to do this is to uniformly convert all primitive data types to ‘normalized’ representations or forms when being transported between machines via the debugging system; the normalized forms are then converted back to the representations native to the individual architectures at both ends.

Of course, this process cannot be completely successful in all cases, as a result of the various reasons described above. Thus, it may be necessary, as appropriate, to view certain primitive data types as completely abstract objects — for example, machine instructions, or even pointer values (memory addresses). However, in view of the fact that ‘address arithmetic’ will often be performed on memory pointers, either manually by the programmer, or automatically in some situations by the debugging system, then it is impractical to routinely consider all (remote) pointer values as abstract objects. Rather, a more realistic solution in this case would be to proceed upwards one additional level of abstraction, corresponding to the communication and interpretation of most all pointer values via simple integer offsets with respect to some previously-communicated and semi-invariant absolute base address value, such that the offsets would take on values only over a relatively small (and therefore easily representable) range. With this extended abstraction scheme, only the base address value would need to be considered as an abstract object, and of course arithmetic on the integer offsets would be a completely straightforward matter.
matter.

In addition, note that insofar as abstraction levels are concerned in general, the ideal debugging system would facilitate manipulation of the abstract data state of a program in terms of the *highest possible forms of abstraction*, as touched upon in section 2.3.1. In constrast, the level of data abstraction discussed so far in this section has corresponded to the relatively low level of architecturally-independent, primitive data types; it is this level of abstraction that in fact established the basis for the experimental implementation of the RXDM remote execution and debugging monitor that will be described in Chapter 4. However, the 'best' abstractions to use, and a corresponding implementation of an advanced, 'abstract debugging system' based on these abstractions, is a topic of ongoing research. Some of the relevant issues involved in this research will be covered in greater detail in the final chapter of this thesis.

The second level of this data portability problem relates to the location (or, even dynamic relocation) of parallel code segments in the context of a distributed inhomogeneous computing environment. One instance of a computation may differ from another simply in terms of where the computation is performed and how the component tasks are distributed — that is, which subset of computing elements constituting the distributed environment is utilized in performing the computation from one instance to the next, and which code segment resides on which element in each case. If the computing environment were homogeneous (all elements alike), then this would not be an especially complicated issue — simple load balancing (either statically or dynamically) could likely be the primary consideration in the allocation of tasks (and code segments) amongst the elements in this case. However, since the computing elements can be different in the inhomogeneous situation, then in addition to load balancing, one must consider that certain distributed tasks may be more appropriately/optimally carried out on one element *versus* another, as a consequence of special characteristics or requirements innate to each individual task. In particular, one such special requirement would be the need to perform numerical calculations at a specific minimum level of precision, which typically would require that the primitive data types involved in the calculation have a certain minimum precision, as well. A candidate computing element that would be a suitable substrate for the computation must therefore support the required level of precision. In addition to this, special consideration must also be given to the likely possibility that typed data of varying precision will be routinely communicated *via* messages passed between the various elements involved in the computation. Therefore, an unavoidable loss in precision will
be experienced whenever data is transported from a higher-precision sending element to a lower-precision receiving element, due to the associated rounding or truncation that occurs. This may or may not be so bad, depending on what is subsequently done with the received data. Certainly, a worse situation results from the reverse scenario — namely, the introduction of low-precision data into a high-precision calculation. A pathological example of this would be the situation in which a low-precision element acts as a forwarding agent for (originally high-precision) data being passed from one high-precision element to another.

This last point is made especially relevant in light of the fact that certain applications can suffer quite seriously as a result of loss of data precision. For example, a great many numerical applications involve algorithms that are very sensitive to the precision of the data involved in the calculations. Consequently, the performance of parallel/distributed forms of such applications in the context of an inhomogeneous distributed computing environment can be crippled unless serious consideration is given to the selection of which specific elements in the distributed environment should participate in the computation. In particular, this selection — which also could be dynamic — should be made both in regards to differences in the internal precision of relevant data types on the individual computing elements, as well as to the overall communications strategy employed in passing precision-sensitive data between different elements. Thus, for example, the programmer or a user might specify that a distributed numerical computation be carried out using a given level of precision, which would then force the distributed computation to be automatically mapped to some subset of machines on the network that specifically satisfy these restrictions *viz.* precision.

In general, any mapping of a parallel/distributed task results in special problems for a debugging system. At the most basic level, the problem simply involves determination of the *location* of a particular code segment, which can be accomplished using the original mapping information. A more complex aspect of the problem, however, results specifically from the inhomogeneous character of the distributed computing environment — namely, that the debugging system will probably wind up 'in the middle of things' with respect to the manipulation of variable-precision data. For example, as part of the normal debugging process, the programmer may utilize the debugging system to manually fetch and re-send data between computational elements having greater innate precision than that (those) on which the debugging system is based. This is equivalent to the debugging system being a low-precision, intermediate
ing computation. Thus, the data transport mechanism of the ideal debugging system ought to be carefully designed so that it does not deteriorate the precision of the data transported between differing computing elements. Furthermore, the ideal debugging system should also facilitate the aforementioned selection process for suitable computing elements, either manually by the programmer or (semi)automatically. In fact, the programmer could utilize such a feature to relocate/redistribute experimentally (perhaps at any time) the tasks involved in a computation, in order (say) to check its performance under differing types of computational criteria (e.g., viz. precision).

2.4.3 Debugging Equivalent Sequential and SMMP Program Code via Reversible Execution

There is a strong mathematical relationship between equivalent specifications for the same input/output transformation, that are targeted for execution on sequential machines versus shared-memory multi-processors (SMMP machines). This relationship is entirely algebraic, and with certain restrictions, is expressible largely in terms of matrices involving multivariate polynomials (reference Rice University Technical Report [...]). That is, an algebraic model representing the sequential version of a computation can be determined, and then an equivalent multi-step SMMP computation can be derived simply by means of application of algebraic manipulations, in particular certain matrix operations, to the sequential model. In fact, in many cases it is possible to derive algebraically an equivalent one-step SMMP computation, if it exists, starting from the sequential model. Furthermore, the process also is invertible, so that equivalent sequential computations can be derived starting with single or multi-step SMMP code.

This model (actually an algebraic model for computation) could have several applications in the area of compiler technology. However, in the context of debugging, it can be used to derive a theorem relating reversibility of execution of sequential code to that for ‘equivalent’ SMMP code — that is, particularly with respect to code that is equivalent under a general class of linear matrix algebra transformations, as defined explicitly in the aforementioned Technical Report. Before proceeding, however, note the following definitions.

Defn. (1): A linear transformation matrix is defined to be a real-valued, square matrix that maps an ‘initial’ data state vector to a ‘final’ data state
Defn. (1): A linear transformation matrix is defined to be a real-valued, square matrix that maps an 'initial' data state vector to a 'final' data state vector, such that each matrix element has no explicit (i.e., symbolic) dependence on any element of the data state vector. In comparison, a nonlinear transformation matrix is defined similarly, except that at least one of its elements will exhibit an explicit, symbolic dependence on elements of the initial data state vector.

Defn. (2): A computation sequence $L^A$ is a sequence of matrix applications that represents a given computation ('A') in this model. That is,

$$L^A = <T_{f_1}, T_{f_2}, \ldots, T_1>,$$

where $\forall j$, $T_j \in R^{n \times n}$ is a real square matrix with dimension $n$ equal to the dimension of the data state undergoing evolution (via individual steps $1$ through $j$) during the computation. Thus, if $D^j \in R^n$ is a real vector representing the data state after step $j$ of the execution sequence, where $D^0$ is the initial data state, then the final data state after completion of the computation is

$$D^A = T^A \ast D^0,$$

where $T^A = T_{f_1} \ast T_{f_2} \ast \ldots \ast T_1$.

Observe that the matrix $T^A$ is the one-step SMMP equivalent of the sequential application of the individual matrices $T_j$.

In addition, a computation sequence is termed linear iff each constituent matrix application is linear (and likewise for the nonlinear case). Similarly, a computation 'A' is termed linear if it is representable by some linear computation sequence $L^A$.

---

24 In fact, even a nonlinear computation can be considered linear once all the symbolic appearances of the initial data state vector elements in the sequence of nonlinear matrix applications have been numerically instantiated.
Defn. (3): A computation is defined as irreversible iff, given the computation and an arbitrary final data state vector, it is not possible to determine uniquely any of the elements of the corresponding initial data state vector that would yield the given final vector after application of the computation. A computation is defined as weakly reversible iff it is possible to determine uniquely from the given information at least one, but not all of the elements of the initial data state vector. Finally, a computation is defined as strongly reversible (or, just reversible) iff it is possible to determine uniquely every element of the corresponding initial data state vector. Likewise, the reversibility rank of a computation is defined to be the number of elements of the initial data state vector that in fact can be determined uniquely. For a nonlinear computation, note that the reversibility rank itself can be a function of the values in the initial data state vector. In such cases, it nevertheless may be possible to formulate domain constraints on the initial data state vector that restrict the circumstances under which the computation is (say) reversible. However, for any given linear computation, its reversibility rank always is a fixed number with respect to variations in initial data state vector values.

2.4.3.1 Conditions for Reversibility of a Computation

Let $\equiv$ be the equivalence relation such that $L^A \equiv L^B$ means

$$D_{\text{final}} = T^A * D^0 = T^B * D^0,$$

and linear sequence $L^A$ is derivable from linear sequence $L^B$ via application of some finite set of linear matrix transformations to the matrices constituting $L^B$ (and vice-versa). Then we have the following theorem concerning reversibility of execution of equivalent (under $\equiv$) computations that are representable by linear computation sequences:

Theorem 2.1: Consider any two linear computation sequences $L^A$ and $L^B$, such that $L^A \equiv L^B$. Then $L^A$ is reversible if and only if $L^B$ is reversible.
Proof: In general, a linear computation sequence $L^Q$ is reversible iff an inverse linear computation sequence $L^{-Q}$ exists. This is the case iff each matrix defining $L^Q$ has a multiplicative inverse; that is, for each step $j$, $T_j^{-1}$ exists such that $I = T_j^{-1} \ast T_j$, and thus

$$L^{-Q} = \langle T_1^{-1}, \ldots, T_{f_{q-1}}^{-1}, T_{f_q}^{-1} \rangle.$$ 

This will be true iff the determinant for each matrix $T_j$, $\text{det}(T_j)$, is non-zero. Now, since $L^A \vdash L^B$, then by definition $T^A = T^B$, so

$$T_{f_{A}}^A \cdots T_{1}^A = T_{f_{B}}^B \cdots T_{1}^B.$$ 

Since the determinant of the product of a sequence of matrices is equal to the product of the determinants of the individual matrices in the sequence, then every $T_j^A$ has a non-zero determinant iff every $T_k^B$ does. But from the definition of reversibility, this implies the result that $L^A$ is reversible iff $L^B$ is reversible. □

This theorem may be specialized to yield the following lemma concerning reversibility of execution for equivalent (under $\vdash$) sequential and SMMP computations:

Lemma 2.2: A linear sequential computation that is representable by linear sequence $L^S$ is reversible if and only if there exists an SMMP computation represented by linear sequence $L^P$, such that $L^P \vdash L^S$ and $L^P$ is reversible.

Proof ($\Rightarrow$): The linear sequential computation $L^S$ corresponds to application of a sequence of component matrices $T_j^A$, each of which incrementally mutates the internal data state in a non-parallel manner (i.e., each step affects only a single data element). Now, the single matrix $T^S = T_{f_{A}}^A \cdots T_{1}^A$ corresponds to an equivalent (under $\vdash$) single-step, linear SMMP computation $L^P = \langle T_{1}^P \rangle$, where $T^P = T_{1}^P = T^S$. Since this $L^P$ is trivially equivalent to $L^S$ under $\vdash$, then this establishes the existence part of the inference. Next, simply let $A = 'S'$ and $B = 'P'$, and apply Theorem 2.1 to establish the result. □
(→): The proof for this follows directly from Theorem 2.1. □

2.4.3.2 Implementation of Reversible Constructs

In this subsection, I would like to explore some of the implementation issues concerning reversible execution for the purpose of debugging. Basically, as discussed above, in order for a linear computation step to be reversible, the determinant of its corresponding matrix must be nonzero. But, in fact, most computation steps correspond to matrices having a zero determinant. Most commonly, this obviously is the case whenever the current contents in a data variable are destroyed as the result of assignment of new contents into that variable — the previous contents are subsequently 'lost', and cannot be inferred at a later time from the resulting data state(s). In terms of the physics of information processing, as promulgated by Bennett (reference [Ben88], among others, this loss in information corresponds to an irreversible ('nonconservative') increase in computational entropy for the system (and universe), in accordance with the second law of thermodynamics.

However, it is indeed possible to (re)design an arbitrary computation in order to make it reversible and thus physically conservative. Bennet explores this matter on a conceptual level. I herein will explore it on a concrete level, at least in terms of the linear computation sequences introduced in this section.

Given an irreversible computational sequence, the general idea is to find a corresponding reversible sequence from which can be derived, under certain circumstances or restrictions, the original computational sequence, on a step-by-step basis. In particular, in accordance with this principle, I will explore herein three explicit methods that may be employed to reformulate the original computation in order to make it reversible. The first of these is the well-known method of storing history information. The second method involves finding (when possible) a reversible computation sequence based on self-referential, incremental changes to the data state at each step, that is computationally equivalent to the original, irreversible sequence under certain circumstances or restrictions on the initial data state. The third method, which I call the branching method, is novel, and consists of splitting the irreversible computation sequence into two or more new, reversible computation sequences that can execute asynchronously in parallel on an SMMP system, individually yielding results that can then be recombined to yield the final result corresponding to
the original computation.

The first method consists of storing history information at appropriate points in the computation sequence. Basically, the idea is, for each data assignment that would otherwise result in the loss of data currently stored in the target variable, to first store (i.e., 'log') the current contents of the target into a separate 'history' variable before performing the destructive assignment. This history variable subsequently will not be reused for any other purpose, besides simply holding the stored contents for later use in accomplishing the reverse computational sequence. Thus, there must be a unique history variable for each such destructive assignment. In the most naive, worst-case form of implementation, this amounts to 'logging' the entire data state vector before each SMMP computational step, since the step may involve multiple simultaneous assignments. However, in reality, only the contents of those specific variables or elements of the data state vector that are being irreversibly modified during that particular step need be logged before the computational step is performed.

Therefore, this history method involves increasing the dimension of the data space by (at least) the actual number of history variables required. But this number is undecidable, since it depends on the actual number of assignments occurring during the overall computation — that is, not only on the number of variables already in the data space, but also on the number of individual computational steps involved in the overall computation sequence. Of course, the latter is dependent on the initial data state (i.e., inputs), which is generally nondeterministic, so the number of required history variables therefore is undecidable for an arbitrary computation.

As a consequence, one could consider having an infinite pool of history variables available for use. For example, one could simply stream the history information onto an infinitely-extensible output 'tape', as mentioned by Bennett, corresponding in concrete terms, say, to a dynamically built, linked list of history data items. This of course will work in the sense that it enables reversibility, but it is not a part of this model! Instead, in terms of the confines of this particular model, all additional storage required for history information must be statically available from the beginning of the computation. Actually, in a sense, this corresponds to reality; namely, all real-world, physical memories are clearly fixed in size. So, whatever the actual extent of physical memory is, one could define that as the dimension of the data space for the computation, and assume/hope that it is not exceeded during the course of storing the history information.
As an example of how the addition of a history variable can enable reversibility, consider the toy SMMP problem (notation: "||" means simultaneous/parallel execution of the statements it separates, while ';’ delimits sequential execution)

\[
\begin{align*}
d1 & \leftarrow 3 \, d2 \ | | \\
d2 & \leftarrow 2 \, d2;
\end{align*}
\]

which has the corresponding matrix

\[
\begin{pmatrix}
0 & 3 \\
0 & 2
\end{pmatrix}
\]

This matrix has a zero determinant, in accordance with the fact that the computational step is irreversible. The computation can be revised, \textit{via} the addition of the new history variable \(d3\), so that it in fact becomes reversible, as follows (assume that, initially, \(d3=0\))

\[
\begin{align*}
d1 & \leftarrow 3 \, d2 + d3 \ | | \\
d2 & \leftarrow 2 \, d2 \ | | \\
d3 & \leftarrow d1;
\end{align*}
\]

which has the corresponding matrix

\[
\begin{pmatrix}
0 & 3 & 1 \\
0 & 2 & 0 \\
1 & 0 & 0
\end{pmatrix}
\]

This new matrix has the nonzero value \(-2\) for its determinant, so it is reversible, with the inverse matrix

\[
\begin{pmatrix}
0 & 0 & 1 \\
0 & \frac{1}{2} & 0 \\
1 & \frac{-3}{2} & 0
\end{pmatrix}
\]

corresponding to the reverse computation

\[
\begin{align*}
d1 & \leftarrow d3 \ | | \\
d2 & \leftarrow \frac{1}{2} \, d2 \ | | \\
d3 & \leftarrow d1 + \frac{-3}{2} \, d2;
\end{align*}
\]
Note that there are several other ways to use the new history variable besides that shown, in order to implement reversibility for this example. However, these alternatives should be fairly clear, so I won't elaborate.

Now, observe that not all computations involving assignments are irreversible. Thus, it is not true that every assignment will require the use of a new history variable to retain destroyed information. In particular, certain computations involving self-referential, incremental changes to variables may be inherently reversible. For instance, the SMMP computation

\[
\begin{align*}
    d1 &\leftarrow d1 + 3 \cdot d2 \\
    d2 &\leftarrow 2 \cdot d2;
\end{align*}
\]

corresponds to the matrix

\[
\begin{pmatrix}
    1 & 3 \\
    0 & 2
\end{pmatrix}
\]

which has the nonzero value 2 for its determinant, implying reversibility via the inverse matrix

\[
\begin{pmatrix}
    1 & -3 \\
    0 & 2
\end{pmatrix}
\]

which corresponds to the reverse computation

\[
\begin{align*}
    d1 &\leftarrow d1 - \frac{3}{2} \cdot d2 \\
    d2 &\leftarrow \frac{1}{2} \cdot d2;
\end{align*}
\]

Furthermore, observe that in this example, if we assume that initially \( d1 = 0 \), then it is in fact computationally equivalent to the first example given above. The conclusion is that, certain irreversible computation sequences can be 'made reversible' by reformulating the offending assignment statements in the computation to be expressed in terms of self-referential, incremental changes. Notably, such a reformulation does not result in the requirement for increasing the dimension of the data space in order to accommodate additional history variables.

Besides these methods, another approach for implementing reversibility is the aforementioned branching method. This method does not normally involve the expli-
cit introduction of new history variables, so the explicit dimension of the data space remains unchanged. Instead, for each irreversible step encountered in the computational sequence, two or more new, reversible computation (sub)sequences are started, and the original computational sequence is correspondingly suspended. These new subsequences may thereafter execute *asynchronously* in parallel; the final results of each can ultimately be recombined to yield the result corresponding to the original computation. Most importantly, though, each subsequence can be reversibly executed, up to the point where it was created as a result of being split off from the original sequence — likewise, by definition, the original sequence can then be reversibly executed from that point on.

More explicitly, the method involves the following. Consider matrix $T_k$, corresponding to the $k^{th}$ step in a linear computation sequence $L$, such that $\det(T_k) = 0$, and furthermore assume that $\det(T_i) \neq 0$, $1 \leq i < k$. That is, this particular step is the first irreversible step in the sequence. Next, consider rewriting $T_k$ as

$$T_k = T_k^L + T_k^R$$

such that each of the matrices on the RHS has a nonzero determinant (*i.e.*, is reversible). For example, one could find a perturbation matrix $H_k$ such that

$$T_k^L = \frac{1}{2} T_k + H_k, \; \det(T_k^L) = 0,$$

$$T_k^R = \frac{1}{2} T_k - H_k, \; \det(T_k^R) = 0.$$  

---

25 Note that the matrix is shown rewritten in terms of two new matrices, but this number is arbitrary, so long as it is two or more.
Thus,

\[
L = < \cdots, T_k, \cdots > \\
= < \cdots, (T_k^L + T_k^R), \cdots > \\
= < < \cdots, T_k^L >, \cdots > + < < \cdots, T_k^R >, \cdots > \\
\text{(by distributive property of matrix multiplication)} \\
= < (L^L + L^R), \cdots >
\]

That is, two new subsequences have been split off, or spawned at the \( k^{th} \) step of the original sequence \( L \) — namely, \( L^L \) and \( L^R \) — whose respective first steps are reversible. Furthermore, this branching process can be continued, as necessary, within each of the new subsequences.

Observe that, after the branch point, the execution of each subsequence can be continued completely independently from that of the other. However, each execution instance will have to be provided with a separate copy of the data state on which to operate, as defined at the end of the \((k-1)^{st}\) step in the original sequence — or rather, at least all those elements of the post-(\(k-1\))\textsuperscript{th} data state that subsequently can be \textit{modified by the other subsequence}. Therefore, although this method does not explicitly increase the \textit{dimension} of the data space as does the history variable method, it nevertheless entails the availability of additional storage for those copies of the data state values necessarily introduced after each branch point.

Thus, this method produces a \textit{reversible computation tree}, with the results at the leaves being combinable to form the final resultant data state corresponding to the original computational sequence. Every path in this tree originating from some interior node can be forward-executed separately from any other path originating from the same node, either in a tightly-coupled or loosely-coupled context. Also, by definition, every (reverse) path from any leaf to the root is amenable to reverse execution. A question of primary interest to the programmer is, which path(s) should be reverse-traced? Also, if multiple paths are involved, how should the information best be presented to the programmer? Furthermore, what specific perturbation matrices should be used at the branch points? \textit{All of these questions, as well as others, are good areas for further research.}

Note that Figure 2.4 illustrates the branching method for the following simple, \textit{contrived} example involving the 2-step SMMP swap of the contents of variables \textit{d1}
and $d_2$, via an intermediate variable $d_3$:

$$d_3 \leftarrow d_1; \quad /* \text{step \#1} */$$
$$d_1 \leftarrow d_2 \parallel$$
$$d_2 \leftarrow d_3; \quad /* \text{step \#2} */$$

Figure 2.4(a) illustrates the two matrices corresponding to these two steps, as well as the equivalent single-step SMMP matrix. (Note the number present at the lower-right
outside corner of each matrix is the value of the determinant for that matrix.) Each of these steps is irreversible, as indicated by the zero value for their respective determinants. Figure 2.4(b) illustrates the use of the branching method to form an equivalent four-leaf/four-path reversible computation tree²⁶, each of whose matrices has a nonzero determinant, indicating that the corresponding computational step is reversible, as required. This figure also shows the equivalent single-step computation matrix for each of the four possible paths through the tree. Note that, if these four matrices are summed together, then the resulting matrix is the same as the original single-step SMMP matrix shown in part (a) of the figure.

2.5 Some Informal Comments on Debugging and Machine Learning (or, Man versus Machine)

If one accepts that the notion of intelligence is embedded in philosophy, then it is naturally by necessity that one discusses machine intelligence from a philosophical point of view. In this regard, I will in this section attempt to explore the connection between the programming/debugging process and the functionality involved in machine learning.

The empirical debugging cycle, as discussed above, is fundamentally related to the feedback learning cycle, as illustrated in Figure 2.5. This figure shows that these two processes differ principally with respect to a simple redefinition of which functional activities in the respective cycle are contained within the user's world as opposed to the machine's world, and also regarding the addition of 'data state' as an expected/produced observable in the case of the debugging cycle (in addition to the normal 'data output'). cycle.

Investigators such as Rumelhart et. al. (see [RHW86a] and [RHW86b]) have demonstrated feedback learning rules for layered neural networks based on the error measured between the output produced by the network and the expected output

²⁶ As previously mentioned, many such trees are possible for any given computation. In fact, the number is unlimited.
Figure 2.5: The Analogous Relationship Between the Debugging and Machine Learning Cycles

(that being taught), given a fixed (teaching) input. The learning cycle in this situation is implemented by a process involving repetitive forward and backwards sweeps through the layers of the network, with the backwards sweep corresponding to an incremental correction to the internal data state of the network in order to improve
the error measure on the subsequent forward sweep. It can be shown on mathematical grounds that this repetitive process can eventually evolve the internal data state of the network so as to effect the correct transformation between the input/output pair being taught.\textsuperscript{27} In fact, the network may in this way be taught multiple input/output pairs. Also, if properly designed, more complex networks can be taught sequences involving temporal relationships between the input/output sets. Making reference to the figure, one can see that the backwards sweep corresponds to the conceptual feedback path through the boxes labelled ‘modify data state’ and ‘modify execution state’, and the effective application of the learning rule corresponds to the actions taken by procedures hidden within these boxes.

The debugging cycle involves basically the same process. The goal is unchanged: to modify the internal data state of the debugged program in order for it to effect the ‘correct’ transformation between the input/output set (including whatever temporal relationships that pertain thereto). Here, however, the program cannot semi-autonomously learn the correct transformation, since it does not have a self-contained implementation of the learning rule mechanism. Instead, of course, the implementation of the ‘learning rule’ is from an external source: namely, the developer of the program (\textit{programmer}). That is, ‘debugging’ is, in effect, a particular form of applied learning rule for the system (the applications program), which is implemented externally by the programmer.

Again making reference to the figure, one sees that for the learning cycle scenario, the ‘modify’ boxes as well as the ‘error computation’ functions are all part of a single, semi-autonomous system (teachable automaton), internal to the machine world and transparant to the programmer/user. In contrast, for the (extreme) debugging cycle scenario, all these functions are external to the machine world and the responsibility of the programmer. The conclusion is this: when the programmer is debugging a program, he is simply in effect carrying out the execution of a negative-feedback learning rule in a process involving directly-coupled interaction with the machine world. (Comment: could one conclude that, in comparison to any other man-machine interaction or activity (such as simply running a program in order to

\textsuperscript{27} which possibly also could involve \textit{temporal} information, as with sequencing
get results), the debugging process actually comes the closest to fulfillment of the term 'melding of mind and machine'?)
Chapter 3

The $\mathbb{R}^n$ Model for Remote Execution and Debugging of Computationally Intensive Programs

3.1 Introduction

The goal of this chapter is twofold: namely, first to present a foundation for basing the development of the existing Rice $\mathbb{R}^n$ program execution and debugging (PED) system, and then to show how this system can be extended, in a natural and logical sort of way, for remote debugging of computationally-intensive programs\(^1\) in an inhomogeneous, distributed computing environment\(^2\), which I term a conglomerate computing environment. Lastly, I proceed to formulate a strategy for implementation of this extended $\mathbb{R}^n$ system, which will culminate in the design and implementation of the RXDM remote execution and debugging monitor presented in the next chapter.

3.2 A Progression of Models for Program Execution and Debugging

In the following discussion, I will attempt to present an outline for the strategies involved in the development and design of various forms of debugging systems. This outline is formulated in terms of a progression of models for program execution and debugging, where the construction of a given PED model is based on the fundamental ‘translate/debug’ stage corresponding to a particular level of abstraction viz. the abstract data state of a program, as described in section 2.3.1. The end product of

---

\(^1\) That is, the subject parallel/distributed application programs are computationally intensive in the sense that they typically are very large, complex, and mostly compute-bound (vs. I/O-bound).

\(^2\) That is, inhomogeneous in the sense that not all the various machine architectures, compilers, and operating systems constituting the distributed computing environment are necessarily alike.
this will be the presentation of a model for the existing Rice R3 system, as well as a revised version thereof that reflects extensions for its operation with remote domains. Also, I will attempt to include in the discussion for each model, a description of some of the positive and negative characteristics pertaining to that particular model.

Note that, in a sense, the 'progression' referred to here basically means the process of shifting the burden of maintaining explicit architectural knowledge to lower and lower levels of abstraction, in order to insulate the upper levels from being sensitive to architectural variations at the lowest, or machine level.

3.2.1 Models for Local Program Execution and Debugging

The models presented in this section will be oriented towards debugging of programs that are locally resident — that is, running on the same machine from which they will be debugged by the programmer.

3.2.1.1 The Conventional Low-Level Machine Language Model (add)

The canonical model for conventional low-level debugging is illustrated in Figure 3.1.3 A good example of a debugging system represented by this model is the well-

---

3 Note the use — not only in this chapter, but throughout the remainder of this thesis — of the following special notation/abbreviations in the figures illustrating a particular PET model:
'A' — 'abstract data state'
'S' — 'specification' (e.g., program source text)
'P' — 'programmer/user'
'T' — 'translate/modify function' (e.g., compilation)
'D' — 'de-translate/inspect function' (e.g., dis-assemble)
'DB' — 'translation database'
'NT' — 'network transport function' (e.g., 'NFS' = Network File System)

In addition, note that, as a rule for each of the models shown, the source text specification 'S' can represent multiple, separately compiled subunits, possibly even written in different programming languages, which are subsequently linked via the generalized 'compilation' process (in the figures, this is part of the functionality of the associated 'T' function) to form the next (lower) level of abstract representation for the program. However, it would in general add little to the understanding of a given
known UNIX *adb*.

This model reflects the high degree of manual interaction involved between the programmer and this PED system. In particular, it indicates that the debugging effort is performed primarily at the level of machine language constructs, and that higher-level debugging in terms of the source code requires the programmer to manually perform the mapping function between low-level machine language representations and higher-level source code representations. In order to accomplish this, the programmer must interact directly with the abstract data state of the program at the machine language level, and manually (mentally) perform the required cross-references between the various abstraction levels using relevant information contained in the symbol tables generated by the compiler and the assembler.

Thus, besides the inconvenience suffered on the part of the programmer by his having to explicitly perform the mapping between the low-level machine language representation of his program and the corresponding high-level source code representation, a potentially severe shortcoming of any system based on this model is that the programmer must have intimate knowledge of the particular architecture for the machine on which his program is to be debugged. This is true for two reasons — first, because his interaction with the program’s abstract data state via the debugger is in terms of machine language constructs; and secondly, because the available cross-reference information involves architecture-dependent memory address references, which the programmer must interpret (for example, as in determining the location of a particular element in a multi-dimensional array of data items).

---

model to somehow reflect this potential multiplicity in the figure illustrating that model, so it typically is not shown explicitly.

4 In the figure, these correspond to the small circles labelled DB, standing for ‘translation Database’. As generically described in section 2.3.1, these are created and updated via the associated ‘translate/modify’ function, labelled ‘T’ in the figure, which in this case correspond to compilation and assembly steps, respectively.
3.2.1.2 The Conventional Source-Level Model (dbx)

The model for conventional source-level debugging is illustrated in Figure 3.2. The motivation behind this model is that, when engaged in the debugging of a program, the programmer ought to be able to interact with the abstract data state of the program at the same level of abstraction that originally was used to specify the program in the first place; namely, in terms of high-level source language constructs. In order to accomplish this goal, the model provides for the debugging system to perform automatically the mapping between the lowest-level representation of the abstract data state, based on concrete machine objects, and a higher-level form of representation based on the source code. In effect, this source-level approach to debugging remedies the problems pertaining to the low-level machine language model that were discussed.

---

5 In particular, this model represents the basic functionality of the ubiquitous UNIX source-level debugging system, dbx.
above.

Figure 3.2: Model for Conventional Source-Level Debugging (e.g., using dbx)

Note that there are several significantly peculiar aspects of this model as shown. In particular, the programmer is assumed to interact with the system in two separate ways or modes, depending on whether he is engaged in program (re)specification or explicit debugging activities; these are illustrated in the figure as the source-level programming environment and the source-level debugging environment abstraction layers, respectively. In regards to the former, the programmer interacts with the source code directly, without going through the debugging system, and initiates compilation as an independent function for translation of source code to machine objects (reference the ‘translate’ function bridging the translation boundary labelled “TB2” in the figure).6

6 Of course, there may be additional, intermediate layers involved in this case, such as a layer corresponding to an intermediate machine language representation, as shown in the previous model.
Regarding the latter mode of interaction with the system, the programmer interacts not with the physical source code itself, but rather with the source-level debugging environment abstraction layer, which contains is based on so-called ‘source-level objects’ defined and supported by the debugging system. These objects basically correspond to debugger-specific abstract representations of the original source code, so the two are in fact closely related. Actually, in a sense, these abstract objects effectively can be considered an extended ‘image’ of the actual source code, that is maintained by the debugging system specifically for the purpose of facilitating debugging interactions with the programmer. The translation boundary (labelled “TBI” in the figure) between this ‘image’ and the run-time environment abstraction layer, which contains the actual low-level, concrete machine objects, is bridged by functions carried out by and corresponding to the source-level debugging system (e.g., dbx). Execution of these functions involves access by the debugging system to the cross-reference information contained in the external, compiler-generated symbol table, as well as to the actual source code text. Typically, the debugger will utilize the compiler-generated symbol table information in order to create its own internally-maintained cross-reference information database. Also, the source text will be utilized in order to generate the ‘source-level objects’-based representation of the program’s abstract data state contained in the source-level debugging environment layer, with which the programmer directly interacts.

In summary then, unlike the previous low-level, machine language-oriented model, this model provides for the debugging system itself (rather than the programmer) to access the cross-reference information contained in the separately-generated symbol table, as necessary and appropriate, in order to effect automatically the required mapping between the low-level, machine objects-based representation of the program’s abstract data state and the high-level, source code-based representation thereof. The result is that the programmer is uncoupled from this mapping function altogether, and no longer needs to interact with the system explicitly at the low level of abstraction inherent in the architecture-dependent cross-reference information contained in these symbol tables. In fact, ideally, the programmer never even sees this information. Thus, source-level debugging not only provides the programmer with the desirable convenience of having a view of his program’s abstract data state that is consistent with the level of abstraction expressed in the original specification of that program, but also (theoretically) insulates him from all the explicitly architecture-dependent aspects of the debugging process.
On the other hand, it is just this automatic interpretation of debugging-oriented information, such as the cross-reference information in the symbol tables, that also can cause a problem. For example, whereas with the previous low-level PED model the programmer was tightly coupled to the cross-reference information contained in the symbol tables, the source-level debugging system now has taken over this role. Thus, instead of the programmer, it is now the debugging system that is totally dependent on explicit knowledge of the detailed representation of this cross-reference information — that is, its extent and notational format. In addition, the mechanisms for accessing debugging information relating to a program, as well as explicitly controlling the execution of that program, is almost always a function of the operating system. Therefore, in general, the design of a good general-purpose source-level debugging system should take into account (besides the usual architecture-dependent considerations) compiler- and operating system-dependent variability in these parameters\(^7\), at least insofar as they are anticipated to occur. The alternative, of course, is to enforce standards for these parameters, to be observed by all compilers/operating systems with which the debugging system will be operated.\(^8\)

### 3.2.1.3 The Basic R^n Model

The model that describes the specific approach taken in the Rice R^n project is illustrated in Figure 3.3. Basically, this model is the same as the previous source-level model, except for the addition of a new abstraction layer, the *interpreteive environment* layer, that contains a representation of the program based on abstract syntax tree (AST) constructs. This layer is included in the model to reflect the possibility of

\(^7\) Of course, one way to facilitate this is to design the corresponding debugging system model to have several stages, so that it has a front-end stage for interfacing to the programmer and a separate back-end stage for interfacing with each disparate compiler/operating system combination. Then in order to accommodate a new compiler/operating system, all that is required is an appropriate back-end stage. Note that this is left implicit in Figure 3.2.

\(^8\) However, even for those instances where this has been attempted, such as with UNIX systems that purport to have a common symbol table definition, there nevertheless can be significant differences in the symbol tables generated by compilers on different machine architectures, which especially could be the case in the conglomerate computing environment.
selective interpretation (execution) of program code fragments as a technique useful in the debugging process, as explored by Chase and Hood (reference [CH87]). The interpretive execution is carried out by the associated 'driver' function shown in the figure, which thus effects a simulated evolution of the abstract data state of the program, for purposes as discussed by Chase. In accordance with the optional nature of these purposes, and as indicated in the figure, this layer can be bypassed dynamically, so that effectively the source-level debugging environment is connected directly with the next abstraction layer, namely the run-time environment. Note that the translation boundary "TB1" separating this layer from the former layer entails a mapping between 'source-level objects' and equivalent abstract syntax. The associated translation database consists of the information required for this mapping, such as cross-references between source statement numbers and pointers to the roots of equivalent AST data structures.

In addition, instead of the generic dbx, a new, custom system for program execution and debugging, called EXMON (for EXecution MONitor), is employed in this model. This new PED tool was developed specifically for research purposes, and in particular was designed to facilitate the efficient implementation of the aforementioned, AST-based selective interpretation feature.

3.2.2 Models for Remote Program Execution and Debugging

Up to this point, the models described have been oriented towards debugging of programs that are locally resident. However, the desired and stated goal is to provide a PED system that will facilitate debugging of remotely resident programs, in particular as relates to parallel/distributed programs.

---

9 One of these purposes, as subsequently elaborated upon in section 5.5.1, is to provide a means for facilitating post-compilation modifications to the program code. That is, modified regions of code, such as dynamically inserted source statements, can easily be executed interpretively, while the unmodified regions, which were previously compiled into machine code, can be executed normally. This scheme effectively bypasses the alternative requirement that some method of incremental recompilation be used to handle the textual modifications. However, there are drawbacks to this scheme, which are discussed in greater detail in the aforementioned section.
Figure 3.3: Program Execution and Debugging Model for the Rice R^n Programming Environment

Clearly, the debugging of parallel/distributed programs involves additional considerations not reflected in these models. Principal among these considerations is the efficient and meaningful communication of debugging-oriented information between the various machines supporting the parallel/distributed computation — that is, efficient in terms of communications delays and overall space requirements for the debugging system, and meaningful in terms of consistency of representation and interpretation of debugging-oriented information irrespective of the specific configuration of the distributed computing environment. Note that this important issue can be complicated in the case of a conglomerate computing environment, by the
existence of architectural differences between machines, as well as by differences in compilers and operating systems.\textsuperscript{10} Similarly, compiler and operating system differences also can result in complications, as previously mentioned in section 3.2.1.2.\textsuperscript{11}

The models presented in this section correspond directly to the source-level and $\mathbb{R}^n$ models described above, to which appropriate alterations have been made in order to facilitate the requisite transportation of debugging information across a new 'network interface' translation boundary separating machines in the (conglomerate) computing environment. Basically, these alterations center about a single issue — namely, how to best distribute the functionality of the debugging system throughout the distributed computing environment\textsuperscript{12}, in order to achieve the desired goal of facilitating efficient transportation of debugging information while insulating the local portion of the distributed debugging system against architecture-, compiler-, and operating system-dependent variations occurring throughout the distributed computing environment.

\textsuperscript{10} For example, the local portion of the debugging system ideally ought to be insulated against architecture-dependent variations in the representation of debugging information with respect to the various remote machines constituting the distributed environment. In particular, the local system ideally should not be required to maintain explicit knowledge concerning the various architectures with which it interacts. Of course, this is the same argument previously brought up in section 3.2.1.2, only with respect to the programmer in that case. There, the goal was to shift the burden of explicit architectural knowledge one level of abstraction down, from the programmer to the (local) debugging system. Similarly, in this case, the related goal is to shift the burden of explicit architectural knowledge one additional level of abstraction down, this time from the local section of the debugging system to the remote section, such that the local section becomes/remains insensitive to architectural variations occurring throughout the distributed computing environment.

\textsuperscript{11} Complications due to differing compiler-dependent representations for symbol table information, as well as for text and data elements that make up a given program, can arise even in an architecturally homogeneous computing environment. Also, different operating systems can provide differing means of accessing debugging-oriented information about programs, as well as differing means for facilitating control of execution of programs.

\textsuperscript{12} A related issue is, where, viz. some relative level of abstraction, to best locate this network interface translation boundary.
3.2.2.1 The dbxtool Remote Source-Level Model

As illustrated in Figure 3.4, one straightforward approach to implementing remote debugging simply is to build a remote communications interface layer (reference the 'remote dbx interface' across translation boundary “TBI” in the figure) on top of an existing standalone source-level debugging system, that facilitates communications between the programmer’s local ‘front-end program debugging environment’, and the remote standalone debugging system’s ‘back-end program debugging environment’ (both of which are based on ‘source-level objects’). In this scheme, the debugging information communicated is basically the same as that communicated between the programmer and the debugging system in the local version of this model (ref. section 3.2.1.2); the difference now is simply that the debugging commands and results entail transportation across the network in passing between the programmer and the debugger!

Note that this approach is in fact the one used in dbxtool. It is actually a comparatively high-level approach, but nevertheless has several drawbacks. One possible drawback is that, because communications of debugging information is performed at a symbolic level, it would be somewhat inefficient if, say, variable names having long lengths were continually passed between machines. Besides this, a much more significant and definite drawback is that, although this approach appears on the surface to provide a straightforward remedy for the architecture-dependent aspects of the problem of remote debugging in a conglomerate computing environment, it is not optimal in the sense that there must be a separately-implemented and entirely complete version of the remotely-located standalone debugging system for each targeted architecture. Also, the overall size of the distributed debugging system will be extremely wasteful of space if the standalone debugging system (which could be rather ‘large’, as in the case of dbx) utilized is normally designed to work with only a single program at a time, therefore requiring that there be an individual instance of the standalone system for each parallel/distributed code segment. Even worse, this approach accomplishes nothing with regard to solving the problems associated with compiler-dependent variations in the representations of debugging-related information, such as symbol tables, as well as operating system-dependent variations in the mechanisms for accessing debugging-related information. That is, the remote version
of the model in itself contributes nothing positive to the solution of this particular problem\textsuperscript{13}, insofar that it exists, as compared to the original local model. Instead, the success of this remote model remains dependent on the original assumption for the local model: that the various, individual remotely-located source-level debugging systems themselves will be able to deal with these variations at some lower level of abstraction. Therefore, in the worst case, there actually must be a separately-implemented and entirely complete version of the remote debugging system for each targeted architecture/operating system/compiler combination possible in the distri-

\textsuperscript{13} This problem is exacerbated by the augmented multiplicity of compilers/operating systems that is possible in the remote/distributed scenario.
buted computing environment.

3.2.2.2 The Extended $\mathbf{R}^n$ Model, as Revised for Remote Domains

The last model to be considered, illustrated in Figure 3.5, is a revised version of the basic $\mathbf{R}^n$ model of section 3.2.1.3 that has been extended for remote program execution and debugging. Note that this model provided the motivation behind the development of the RXDM remote execution and debugging monitor to be described in Chapter 4.

Besides the obvious presence of an AST-based interpretive environment, the basic difference between this model and the previous model mostly has to do with the particular choice of abstraction level for locating the network interface translation boundary. In the previous model, the location of this translation boundary was at a relatively high level of abstraction, with the debugging system 'core' (e.g., dbx) located on the remote machine, 'below' this boundary. In the current model, however, the situation effectively is reversed, with the relative positions of this boundary and the debugging system core (which includes the $\mathbf{R}^n$ EXMON Execution Monitor) inverted as compared to the previous model, such that the core remains local to the programmer, 'above' this boundary.\(^{14}\) Note that one of the reasons for using this different approach is to redistribute the functionality of the debugging system so as to be more heavily weighted towards the local machine than in the previous model, in order thereby to remediate the aforementioned concern about having multiple, relatively 'large' individual instances of the configuration-sensitive, standalone debugging system (e.g., dbx) on each remote machine. Another reason is to facilitate the possibility of more efficient communication of debugging-oriented information between

---

\(^{14}\) In consideration of the original, local versions of these two models, therefore, the question is simply one of where the network interface translation boundary should be located in order to extend the local model for the remote case. For the basic source-level model of section 3.2.1.2, the decision was to locate it towards the programmer's side of the existing translation boundary labelled "TB1" in Figure 3.2, above the debugging system core, yielding the new, remote dbztool model of the previous section 3.2.2.1. For the basic $\mathbf{R}^n$ model of section 3.2.1.3, on the other hand, the decision is to locate it towards the machine's side of the existing translation boundary labelled "TB2" in Figure 3.3, below the debugging system core, yielding the current, remote $\mathbf{R}^n$ model.
machines by foregoing the use of (possibly lengthy) variable names, etc.

Basically, this scheme is effected by first introducing a new abstraction layer (i.e., transport environment, not shown in the figure) between the existing source-level debugging environment and the run-time environment layers, whose purpose is to insulate the former from the architecture-dependent characteristics of the latter. Next, this new layer is split further into two new sublayers by the introduction of the
network interface translation boundary (reference "TB3" in Figure 3.5), with one sub-layer on either side of this boundary, corresponding to a new 'front-end' and 'back-end' environment.

These two new transport environments each contain equivalent representations of the program's abstract data state, as based on constructs from a new 'transport abstraction language', having an 'intermediate' level of abstraction considered suitable for relatively efficient communication of debugging-oriented information between machines. Effectively, the front-end transport environment serves as a local interface between the higher level abstract representation consisting of 'source-level objects', and the new intermediate-level representation, consisting of 'transport objects' that are communicated over the network with the remote machine. Likewise, the back-end transport environment serves as a remote interface between a representation consisting of these communicated transport objects and the low-level representation consisting of concrete 'machine objects'.

Note that, to some degree, this scheme apparently succeeds in implementing the aforementioned goal of more evenly distributing the functionality of the existing debugging system than was accomplished in the case of the model for dbztool, as well as the goal of lowering the level of abstraction of the communicated debugging information in order to increase overall efficiency. However, it nevertheless fails in one very important respect, which the previous model was successful in accomplishing. Namely, as a result of the dependence retained by the original EXMON Monitor on explicit (remotely-generated) symbol table information, this model compromises the fundamental goal of insulating the local debugging system from architectural variations among remote machines constituting the conglomerate computing environment.

---

15 Actually, the concrete agents effecting these interfaces are the so-called 'Remote Monitor Front-End Interface' and the 'Remote Monitor', respectively, as shown in Figure 3.5. The latter corresponds to the concrete RXDM remote execution and debugging monitor to be described in Chapter 4. These will be discussed in section 3.3 as well.

16 This is indicated by the information feedback path connecting the remote symbol table (DB on translation boundary "TB3" in Figure 3.5) to the translation database maintained locally within EXMON (DB on "TB2" in the figure). In contrast, in the previous model of section 3.2.2.1, both of these translation databases were resident on the same (remote) machine.
Therefore, because of this characteristic, this model does not successfully provide for redressing the problem (insofar at it exists) of compiler-dependent variations in the representations of debugging-related information, such as is found in the symbol table. However, this model does do a better job than the previous model in facilitating interfacing with differing remote operating systems, since the (relatively low-level) Remote Monitor probably would be easier to reconfigure for a different operating system than would the 'fixed' standalone debugging system!

3.3 Implementing a System Based on the Remotely-Extended R^n Model

In this section, I will discuss basic issues relating to the implementation of a centralized system for remote program execution and debugging, based on the remote R^n model discussed in section 3.2.2.2. This will then lead to a corresponding concrete design, namely the RXDM remote execution and debugging monitor, to be discussed subsequently in Chapter 4.

Basically, the model of section 3.2.2.2 (reference Figure 3.5) is divided into two portions or sections: one 'local' and another 'remote'. Likewise, any realization of a debugging system based on this model will also have 'local' and 'remote' sections. Note that, whereas the former corresponds to the singular 'core' of a centralized, remote debugging scheme, the latter can correspond to multiple physical instances resident on the various machines constituting the distributed computing environment. Figure 3.6 illustrates the centralized character of the structure of such a remote debugging system.

Now, the model assumes in particular that the local debugging system core is centered around the existing R^n EXMON Execution Monitor, although perhaps with some (relatively minor) modifications, as appropriate, for interfacing with the new remote section of the system. On the other hand, the model introduces two new abstraction layers: namely, the *front-end environment* and *back-end environment*

---

17 There even could be multiple instances on a given single remote machine.
layers. The first of these is local, and is bounded on one side by a translation boundary (reference "TB3" in Figure 3.5) separating it from the source-level debugging environment, and on the other side by the 'network interface' translation boundary ("TB4" in the figure), separating it from the remote section of the PED system. The back-end environment layer is remotely located, and is bounded on one side by the 'network interface' translation boundary, and on the other side by the 'machine ob-
jects' translation boundary ('TB4' in the figure). As discussed in section 3.2.2.2, these new abstraction layers are based on constructs from a 'transport abstraction language'. The purpose of this 'language' is to facilitate efficient and meaningful communication of debugging-oriented information at a level of abstraction that, as much as possible, tends to insulate the local debugging system core from configuration-dependent variations (i.e., viz. differing architectures, compilers, and operating systems) in the remote portion of the system.18

In accordance with the structure of this model, a realization of the remote section can be seen as consisting of two components. One is of course the physical, remote engine or 'machine' corresponding to the realization of the model's run-time environment. The other is the so-called 'Remote Monitor' module, which basically corresponds to a realization of the back-end environment. Note that the RXDM remote execution and debugging monitor, which will be described in Chapter 4, is such a concrete implementation of this Remote Monitor.

Correspondingly, the local core section also can be seen as consisting of two components. The first is of course the existing (though modified) EXMON Monitor. The second is a 'Remote Monitor Front-End' interface module, which basically corresponds to a realization of the front-end environment, and which couples the EXMON Monitor to the Remote Monitor across the network interface boundary. Again, note that the RXDM.FREND module, which also will be described in Chapter 4, is a concrete implementation of this Remote Monitor Front-End, targeted specifically for interfacing to RXDM.

---

18 Actually, however, as discussed in section 3.2.2.2, this model inherently compromises this objective, mainly because of the nature of the existing EXMON Execution Monitor on which the model is based!
Chapter 4

The RXDM Remote Execution and Debugging Monitor

4.1 Introduction

RXDM is an implementation of the 'Remote Monitor' described in sections 3.2.2.2 and 3.3, basically (though not exclusively) for application within the Rice R⁶ programming environment.¹ The primary function of RXDM is the monitoring and control of user applications programs under its supervision (ref. the acronym SAP, for Supervised Applications Program, appearing throughout the following sections) including special features for debugging such as setting and monitoring program execution state, text and data, and breakpoints. In addition, RXDM provides capabilities for facilitating data communications between the user and the applications programs, as well as between the user and the local file system. A significant, related feature is that, in the RXDM view, applications programs may be specially abstracted so that both the program text space and the data space are treated as 'files' (separately), allowing the same set of file operations as are provided by the monitor for any other 'normal' file on the local file system; namely, open, position, read, write, and close. In this way, multiple files may be opened on the 'device' that consists of a particular program data space, each file defined to overlay (say) a certain program data structure, or even the program stack area (e.g., for access to call frames). Also, RXDM file operations allow typed read/write operations, which in this example facilitates accessing arrays of data items of a given type (e.g., integer, floating point, etc.). This scheme likewise permits accessing the program text space via files opened to overlay arrays of instructions, which is an important debugging tool in that it facilitates

¹ However, its design is actually general enough for it to be utilized in other applications involving remote debugging systems, as well. In any such application, all that is required in order to interact with RXDM and utilize the remote debugging features that it provides, is the provision of an appropriate interface module. (See section 4.4 for a description of the particular interface constructed specifically for use with R⁶, known as RXDM_FREND.)
dynamic manipulation (i.e., insertion, modification, and deletion) of entire text fragments as contiguous units. Such fragments could be used to aid the debugging process.

The RXDM monitor is designed in accordance with the model delineated in Chapter 3. That is, it is designed for operation in a conglomerate (i.e., inhomogeneous, distributed) computing environment, and provides features that facilitate the remediation of those special problems or concerns which arise in conjunction with this type of an environment, as discussed in that chapter.

4.2 General Description

The purpose of this section is to provide an introduction to the principles and philosophies involved in the design and operation of the RXDM monitor. The discussion is purposefully maintained at a topical level, without entering into extensive details that would undoubtedly cloud the reader's understanding and appreciation for the issues presented.

Note that name references appearing in capital letters (e.g., \texttt{START\_PROG\_TRACED}) correspond to predefined name constants that are part of the RXDM system, and may be referenced by the user in his code via the appropriate C-preprocessor `#include' directives.

4.2.1 Organization of the Monitor

The RXDM monitor is a shell-type process (in the sense of UNIX shells), whose execution is initiated by the (remote) user process. The monitor is not a formal UNIX server process, nor is it part of a remote procedure call (RPC) mechanism, although in many aspects of its functionality it displays/emulates characteristics of both. There is an exclusive relationship between a particular monitor process and the user that invoked it, although the user may invoke several monitors for 'simultaneous' execution on any one machine in a distributed network. Furthermore, although the monitor responds on demand to a standard set of directives available for issuance by the user (RPC-like behavior), it is also designed to be actively involved in quasi-real time supervision of any user-specified applications programs (SAP's) running under its
care. The monitor will only idle in background awaiting arrival of user requests in case that it has no active SAP's.

Figure 4.1: Functional Organization of the RXDM Monitor

The monitor, which is written in C, comprises several basic modules, including: the main module for execution control/sequencing; various interrupt-handling
modules (viz. system-delivered signals); modules for implementation of high-level (multilayer) user/monitor communications; a process control module for interfacing to the user's SAP's (low-level SAP execution control and monitoring functions, SAP address and state space access functions); a low-level system functions interface module (e.g., calls to operating system primitives), including basic file system interface routines (e.g., I/O byte pumps for network communications links); a low-level system signals interface module; modules for interpretation of monitor directives issued by the user (commands and service requests — see following description); a module for handling SAP execution state change events as well as asynchronous events corresponding to user/monitor data communications with SAP's; a module for implementation of special monitor file abstractions; a module for implementation of local breakpoints; and, several modules for miscellaneous definitions and utility functions.

Figure 4.1 illustrates, in a general, condensed block form, the organization of RXDM with respect to its major functional features. Note that, with certain exceptions as indicated, the blocks with a 'G' enclosed represent some sort of selection/deselection ('gating') processes that are dependent on the monitor execution control and scheduling mechanism (which would also have been indicated as a separate block in the figure, along with the interconnections to the various 'G' blocks, if space had permitted).

Execution of the monitor is driven both by signal interrupts as well as by polling constructs. The main control section is described by the following condensed (pseudo-C) code. Note that each SAP under the care of the monitor is identified by its associated progindex, a small integer into a global table in which the monitor maintains all requisite information pertaining to that SAP.

This code concerns a fundamental set of tasks that form the nucleus of operation of the RXDM monitor. These tasks are:

- interpretation/execution of user-supplied directives (in particular, commands — see the discussion in section 4.2.2), as dynamically stored in FIFO execution queues by a separate interrupt-driven communications handler
- monitoring for SAP execution state changes (running to not running)
- monitoring for production of output data by an SAP (to be forwarded to the user).
main ( ) {
initialize monitor...perform startup handshaking sequence with user;
enable handling of signal interrupts;
    /* This is the main for-loop (execution control driver). */
for (rotation=max(1, noprogs), progindex=0; progindex=(progindex + 1) mod rotation) {
    /* Deal with any asynchronous SAP event activities for this progindex. */
    if (Prog_Active(progindex) status set) {
        /* Check for data on SAP output channel(s). */
        if (Prog_Running(progindex) status set) {
            if (any data present on program output channel(s) for this progindex) {
                fetch data on the channel(s) and forward to user (or file, if redirected);
            } /* if */
        } /* if */
    } /* if */
    /* Perform nonblocking poll to test for any SAP execution state-change events. */
    if (any program has stopped running) {
        let pid = the system process id no. for this program;
        let progins = the RXDM progindex for the SAP having this pid;
        if (the previous state of the SAP with this progins was running or
            (the previous state of the SAP with this progins was paused and
            the new state is terminated)) {
            record state-change and report event to user, referenced by progins;
        } else {
            ignore the event;
        } /* if...else... */
    } /* if */
    /* Handle any commands stored in the FIFO command execution queues. */
    disable handling of signal interrupts; /* necessary for reliable test !!! */
    if (any execution queue is nonempty) {
        re-enable handling of signal interrupts;
        if (monitor execution queue is nonempty) {
            initiate execution of the next pending command on the monitor execution queue;
        } /* if */
        if (execution queue for SAP with progindex is nonempty) {
            initiate execution of the next pending command on this SAP's execution queue;
        } /* if */
    } else
    /* Pause monitor's execution in order to avoid hard-loop busy-wait. */
    { 
        if (any SAP is active) {
            set up for enabling of interrupt handling for any SAP state-change events;
            atomically enable handling of previously disabled signal interrupts and
            implement short delay-pause (approx. 80 milliseconds);
        } else {
            disable handling of interrupts due to state-change of any SAP, as enabled above;
        } /* if...else... */
    } /* if...else... */
} /* for */
} /* main */
The for-loop in lines 4 thru 44 is repeatedly executed, until the monitor terminates, performing the indicated operations as necessary for each active (although not necessarily running) SAP, as referenced by the program number, or \textit{progindex}. However, the individual iterations through the loop are not necessarily intended to correspond to work pertaining \textit{only} to that particular SAP referenced by the value of \textit{progindex} associated with the given iteration. Rather, the loop’s purpose is largely to perform repetitive, nondeterministic polling. Consequently, the apparent design inconsistency in lines 11 thru 21 involving polled-checks for \textit{any} SAP’s while in an iteration apparently dedicated to a \textit{single} SAP (as referenced in that iteration by \textit{progindex}), is reconciled by virtue of the requirement that the checks take as a guard that at least one SAP be \textit{active}, and this guard is supplied for ‘free’ by locating the checks within the if-block of lines 5 thru 22, as shown.

If this loop were free-running, then the monitor process could waste a lot of CPU time just spinning, waiting to act either upon input directives from the user, or upon detected SAP state-change events. Consequently, the code is designed to implement a dynamic, three-tiered scheme for the control-flow connected with execution of these fundamental tasks, according to the relative demands or load being placed on the monitor at any given time. This multi-tiered approach is taken in order to strike a balance between responsiveness of the monitor and the correspondingly proportional demand placed on system resources (i.e., CPU time) by a possibly unnecessarily overactive monitor process.

The first tier (\textit{execution mode}) corresponds to the monitor being in a continual, hard-looping state in order to operate at a peak level (\textit{viz.} responsiveness) suitable for servicing a high load condition, and consequently results in the greatest demand on system resources (CPU time) by the monitor process. As implemented, this mode results in the situation in which there are any pending \textit{command messages} present and awaiting execution on any one of the FIFO \textit{command execution queues} (there is one such queue associated with each active SAP, as well as one associated with the monitor itself). This mode is based on the straightforward goal that the monitor always attempt to initiate interpretation/execution of pending commands whenever it can, but also has its roots in the assumption that the number of commands queued up awaiting execution is a relative measure of load on the monitor. That is, if the user is sending large numbers of commands to the monitor at a high rate (the basis for the definition of ‘high load’), they will tend to pile up in the execution queues. Of course, this simple assumption would be even more accurate if additional information
were taken into account, such as the actual number (or even kinds) of pending commands present on the queues, so that this hard-looping mode would not be entered until absolutely necessary. However, such modifications were not implemented in the current design of the monitor primarily on the basis of their complexity, since it was unclear whether the aforementioned, simple assumption, which is certainly conservative in that it always results in the hard-looping mode at least whenever actually necessary, indeed fails to provide a reasonable minimization of the monitor's demands on system resources. Such a judgement will of course require additional experimental evidence in the use of RXDM.

Note that, in this mode, the SAP command execution queues are checked in turn, sequentially by progindex, one for each iteration through the main for-loop in the above code (round-robin). However, the command execution queue associated with the monitor is checked on every pass through the loop. This is done in order to artificially elevate the relative execution priority of commands targeted for the monitor vs. those targeted for SAP's.

Of course, in this mode, the other monitor tasks are carried out as well. Each hard-loop iteration of the main for-loop entails a (single, nonblocking) scanning/polling operation to implement the monitoring tasks connected with the SAP whose progindex is selected in that iteration. Also, the arrival of certain signals causes the normal monitor execution to be temporarily interrupted in order to process the given signal events. Subsequently, normal execution control resumes at the point of interruption. These signals correspond to:

- the arrival of an incoming directive message from the user (if the directive is a service request — as described in the following section — it is immediately executed within the monitor communications interrupt-handling routine; otherwise, the directive is a command, and is enqueued onto the appropriate execution queue for later interpretation/execution by the monitor; ref. UNIX SIGIO signal)

- breaking of a communications link with the user or with one of the SAP's (ref. UNIX SIGPIPE signal)

- the occurrence of a monitor clock tick (the monitor is provided with a software clock, which generates 'ticks' — under most operating circumstances — approx. every 200 milliseconds; ref. UNIX SIGALRM signal).
The second tier (monitor mode) corresponds to the monitor's execution being temporarily suspended for a short delay time interval (approx. 80 milliseconds) between each iteration through the main for-loop. This mode results from the situation in which there is at least one SAP active (although not necessarily running), but in which there are no pending command messages present on any of the command execution queues (ref. lines 34 thru 38). This condition corresponds to the requirement that the monitor continue to perform scanning/polling operations on a regular basis, in order to detect either a change in an SAP's execution status or the presence of data on one of the SAP data output channels, and then report such event(s) to the user as quickly as is 'reasonable', all without placing an undue burden on system resources (CPU time).

During the time the monitor's execution is not being delayed (blocked), it is sensitized to handle the same set of signal interrupts as described above for execution mode. During the delay period in which the monitor process is being blocked from running, however, it is nevertheless possible for the arrival of certain signals to abort the remainder of the delay period, causing the monitor process to immediately (asynchronously) awaken and continue with the handling of the interrupt event, and then continue with the next iteration through the for-loop. These signals correspond to:

- the arrival of an incoming directive message from the user (ref. UNIX SIGIO signal)
- breaking of a communications link with the user or with one of the SAP's (ref. UNIX SIGPIPE signal)
- a special signal indicating that an SAP has changed state (i.e., under UNIX, the SAP's are child processes of the parent monitor process, and the relevant signal is the SIGCHLD signal).

The third tier (paused mode) corresponds to the monitor's execution being suspended ('paused') for an indefinite (although bounded) time period. Since in this mode the monitor process is blocked by the system from executing, it obviously places the minimum burden on system resources. This mode results whenever there are no pending commands present on any of the execution queues, and there are no active SAP's that require monitoring; that is, basically, when there is nothing for the monitor to do (ref. lines 39 thru 41).

As with the above monitor mode, it is possible for the arrival of certain signals to abort the pause, causing the monitor to awaken and continue. These signals are
the same as previously described for the monitor mode, except that in this case, since there are no active SAP's, it is possible for a UNIX SIGPIPE signal to occur only as a result of the breaking of a communications link with the user.

Observe that, in the case of the monitor and pause modes, if the monitor continues execution from an aborted pause as the result of an interrupt, it resumes executing the next iteration of the main for-loop. It may subsequently (re)enter any of the above three operating modes, as appropriate. Therefore, the dynamic nature of the monitor's control-flow mode selection is ensured.

Also note that all interrupts were by necessity disabled at line 23 before the testing of the execution queues for the presence of stored and pending user commands in line 24. This is because of the following pathological scenario that otherwise would result in the monitor ignoring commands from the user for an unnecessarily long time period (i.e., possibly as long as one monitor clock period, or about 200 milliseconds). Namely, consider the situation in which the interrupts are not disabled at line 23, there are no active SAP's, and all the execution queues are empty at the time the tests in lines 24 and 34 are carried out. The test in line 24 is therefore negative, so execution continues with the test at line 34, which also is negative. Assume, however, that immediately after this last test is carried out, a new user command arrives, which causes execution of the main driver to be temporarily suspended while the command is placed into an execution queue by the monitor's communications interrupt handler routine. After this, execution of the main driver continues at line 40, causing an 'indefinite' pause of the monitor's execution (i.e., the monitor process gets blocked), even though there is now a pending command in an execution queue awaiting attention from the monitor! The monitor will subsequently remain in this blocked state until the arrival of another interrupt causes it to become 'unstuck'. In the worst case, this would be the next monitor clock tick, so the stored user command could remain ignored for an inordinate period of time (with the phenomenon being pretty much a random effect from the user's standpoint)! Furthermore, in addition to this required disabling of all interrupts at line 23, the disabled interrupts must be atomically re-enabled at lines 36, 37 and 40, 41; otherwise, interrupts would be enabled for a (short) time immediately preceding the actual implementation of the blocking pause, thereby providing a window of opportunity for the pathological scenario just described to occur!
Figure 4.2 illustrates the relationships between these three operating modes in terms of the relative execution timing for individual iterations of the main for-loop.

**Figure 4.2: Timing Diagrams for Monitor Control-Flow Operating Modes**
Vertical bars indicate iterations of main for-loop corresponding to given `progindex`, with relative time periods between iterations as shown (time axes not to scale).

4.2.2 Monitor Directives: Commands and Service Requests

RXDM accepts and acts on *directives* issued by the user. Directives are communicated in the form of messages (more explicitly, *call* messages), which contain the requisite information for RXDM to take the action desired by the user. In each case, after acting on the directive, the monitor always responds to the user with a corresponding
reply message, which indicates its relative success or failure in carrying out the specified action. (Therefore, it would seem that communications between the user and the monitor is basically of the 'call-reply' variety; however, there are additional complexities, which will be elaborated in the following discussion. Also refer to Figure 4.3.)

The user presents the directives to RXDM in the form of message packets, which must be specifically constructed in accordance with a predefined set of specifications. That is, each kind of directive has an associated set of standardized specifications for the interpretation of the format of the information in its message packet, whereby the particular format used depends on the context in which the packet is communicated. Namely, for a given directive, there are separate formats for the incoming call message packet from the user as well as for the outgoing reply packet(s) from the monitor. (In fact, for the latter, the particular format used will depend on the relative success or failure on the part of RXDM in executing the directive. It is generally true that, in the case of failure, reply message packets will be 'truncated' by the monitor to contain little or no information beyond the minimum required to provide a clear indication of the problem.)

Directives issued by the user fall into two categories, namely commands and service requests. The fundamental difference between the two is that the latter invoke the immediate attention of the monitor for execution, whereas the former are always placed in one of several first-in first-out (FIFO) execution queues for subsequent, 'scheduled' execution. There is one such queue for each of the applications programs running under the supervision of the monitor, as well as a separate queue for the monitor itself. (Note that there is also another queue, the reply queue, on which the monitor enqueues replies to certain commands it has already executed (see below).)

Commands are placed into the execution queues according to the indigenous operative target associated with the command. Depending on the specific command, this target is either one of the (extant) supervised applications programs (for brevity, SAP's), or the monitor itself. The indigenous target of certain commands is always the monitor, whereas for the others it is always one of the SAP's. Thus, RXDM can infer from the command itself whether to place the command into the monitor queue or into one of the queues corresponding to an SAP (as further identified in the command message packet by a separate field).
If the indigenous target of a command is a particular SAP, then the user must of course identify it to the monitor in the command message packet. For example, when an SAP first is to be started, the user issues a `START_PROG_x (x = NOTRACED, TRACED, ATTACHED)` command to the monitor (which is actually the target of this particular command) that identifies the SAP by means of the file system path name of its executable, which is stored in a string field within the command message packet. After the monitor successfully starts up the SAP, it then assigns it a unique identification number — its program number, or `progindex` — that is returned to the user in an integer field within the reply message packet. Thereafter, all references to the SAP are through this associated `progindex`. Specifically, all subsequent commands having this SAP as their target will refer to it by means of its corresponding `progindex`, rather than by its path name. This approach facilitates efficient handling of commands from the standpoint of communications (shorter message packet lengths), as well as in regards to the interpretation of the command target by the monitor.

All commands may be designated by the user as being either `blocking` or `non-blocking`, by means of a particular control bit in the command message packet. The difference between these two cases is twofold. First, for commands of the nonblocking form, the monitor will always respond with a special `release-reply` message packet *immediately upon receiving the command message packet from the user*, before the command is even queued for execution. This release-reply message packet is typically very short, basically consisting of an echo of certain fundamental information contained in the command message, in particular the original message packet sequence number and command code. Its purpose is to provide the user with an immediate, synchronous confirmation of receipt by the monitor of the command message previous to its actual execution, thereby enabling the user to proceed with other tasks, as desired, before fetching the ensuing asynchronous reply message for the command.

The second difference between nonblocking and blocking forms of commands is that, depending on the command, nonblocking commands are always given a relatively lower priority whenever a scheduling decision occurs during the execution process for the command. For example, the execution process for several commands is more complex than simple fetch-and-interpret, involving in some cases multiple stages of macro-like expansion into other commands. These expansions must themselves be placed onto an appropriate execution queue. The expansion of the blocking form of a command almost always gets pushed directly back onto the top of the queue for (relatively) immediate subsequent execution, whereas the expansion for the
nonblocking form of a command gets placed at the bottom of the queue, behind any other pending commands on that queue, as if the expansion had itself been received from the user. The effect of this strategy on execution time for a command will of course be most pronounced in the situation in which the monitor is heavily loaded in the sense of high activity *viz.* communication, command execution, and many SAP's. For still other commands (notably the *READ_FILE* command), the reply by the monitor consists of a (possibly large) sequentially-generated group of message packets. Specifying the nonblocking *vs.* blocking form of the command in these cases will affect the *rate* at which individual reply message packets making up the reply-group are transmitted to the user (the former will always result in a slower rate of transmission than the latter). This feature may be desirable on the part of the user, who has the responsibility, after all, of *dealing* with this group of rapidly-arriving reply message packets (presumably *via* some queueing mechanism of its own), and who therefore may desire a more 'regulated' rate of their arrival! Therefore, specifying the nonblocking form of a command is actually an indication on the part of the user that a (reasonable) delay in completion of execution of the command by the monitor is not only expected, but is, quite possibly, to some degree even desirable.

The last significant difference between the nonblocking and blocking forms of commands is that the monitor's replies to most nonblocking commands are enqueued for later, 'blocked-data' transmission, in order to increase overall communications efficiency. This queue, the *reply queue*, is periodically flushed by the monitor, such that all of the enqueued messages are sent as a collection of groups of individual messages, whereby each group is a large packet consisting of a contiguous block of bytes corresponding to several complete messages. However, the replies to some commands — notably the *START_PROG_TRACED* command, which is used to start an SAP in a mode suitable for subsequent execution tracing — are never queued in this manner, even if the command were specified in the nonblocking form.

On the other hand, reply messages for certain commands initiated in the blocking form can be *forced* onto the reply queue at the discretion of the user, who indicates this objective as an option in the original command message. In this way, the user may take advantage of the relatively higher execution priority that the blocking form of the command provides, while effectively simulating the nonblocking form of the command in terms of the reply response. In particular, this option is available for the blocking form of the *WRITE_FILE* command, and is useful when there is a large number of writes being sent to the monitor and it is desired to minimize the
inefficiencies inherent in the 'send-and-wait' aspect of the usual call-reply message passing scheme.

Note that there is yet another option available for many commands that specifies that the monitor is *not* to issue *any* reply for the command if the command executes successfully, but rather *only* in the case that it fails. This option is available for those commands that aren't designed to return much meaningful information anyway, but for which the primary purpose of the reply message is simply to provide the user with an indication of completion of command execution. (A good example of this type of command is the **SINGLSTP** command.) The user may find this option attractive when large numbers of these commands are being sent to the monitor and excessive communications overhead becomes an issue. In every case, however, the monitor will always generate the immediate release-reply message for nonblocking commands (which serves to acknowledge their initial receipt), even if this 'no-reply' option has been specified.

As mentioned above, service request directives are given immediate attention upon arrival for execution by the monitor. In general, service requests are 'low-level'-type directives, which always have the monitor as their indigenous target, and which by their nature require high-priority servicing without the scheduling uncertainties/delays involved in execution queueing. Furthermore, service requests may *not* be designated by the user as *nonblocking* — they are always considered to be in the *blocking* form, and an attempt to specify the nonblocking form is ignored. Also, service request reply messages are never placed on the reply queue.

### 4.2.3 Detection and Reporting of Events: Asynchronous Monitor Event Messages

As described above, **RXDM** accepts and acts on specific directives given it by the user, issuing replies back to the user as it completes each directive. However, **RXDM** can also issue message packets asynchronously to the user that are unrelated to any previously received directive. This will happen in the following situations:

- an applications program being supervised by the monitor changes state (with the exception of 'not running' to 'running') — for example, as in the situation that a traced SAP stops/pauses as a result of having encountered a breakpoint
• output data generated by such an applications program has appeared on a monitored data channel (e.g., program stdout), and the monitor is forwarding this data to the user

• the RXDM monitor has detected that it has committed an internal error

• the RXDM monitor has received a request from another user’s monitor for establishment of a communications connection between itself and that monitor (user) for the purpose of ‘talk’ between the users (this situation will be discussed in detail in section 4.3.3, so it will not be elaborated in this section).

• incoming data has appeared on the ‘talk’ communications connection, which was previously configured for asynchronous data delivery

• a ‘talk’ connection, which was previously configured for asynchronous data delivery, has been anomalously disconnected.

These message packets are referred to as event message packets. They are never queued by the monitor for later transmission to the user, as in the case of reply messages corresponding to non-blocking commands, but are always transmitted as quickly as possible. Each event message packet is assigned a sequence number unique from the standpoint of the monitor (but that of course may conflict with sequence numbers originating from the user), as well as a code identifying the type of event (similar to the code for a command or service request) to which the message corresponds, as delineated above. Note that the monitor never expects a subsequent reply or confirmation of receipt of the event message back from the user. Figure 4.3 illustrates communications of these asynchronous event messages, as well as the call-reply and released call-reply communication schemes discussed previously in section 4.2.2, between the user and RXDM.

A primary function of RXDM is to continually monitor the execution status of all supervised applications programs (SAP’s) under its care. If an SAP changes state from ‘running’ to ‘not running’, or from ‘stopped/paused’ to ‘terminated’, then RXDM will automatically detect this change and asynchronously issue a corresponding informational program event message packet to the user. This message packet contains information relevant to the program state change such as its identifying progindex, its final run-state (stopped/paused, exited, terminated), the UNIX signal number that initiated the state change, and the address at which the program halted as well as (in the case of a stop/pause) the machine instruction located at that address. The monitor
also automatically detects if the SAP halted due to encountering a 'remote' breakpoint (which is simply a reserved illegal instruction from the machine instruction set placed in the program at a specified location explicitly via a previous POKE command from the user). If so, this fact is reported to the user in the event message packet in addition to the above information. Likewise, the monitor will detect if the SAP halted due to a so-called 'local' breakpoint (also an illegal instruction, but set by
the user via a previous \texttt{SET_BRKPT} command; see section 4.2.7 on controlling the data state of an SAP), in which case it modifies the aforementioned set of information returned in order to inform the user of the identification of the local breakpoint (i.e., a "\texttt{<fileindex, bpindex>}" pair — see section 4.2.7).

This state-change monitoring applies preferentially to situations in which a state change by a SAP is either expected or unexpected. Clearly, a state change is always expected in response to directives that have this change as their goal. For all such directives, RXDM does not issue a corresponding reply message (given no other gross errors were detected) until after the state change has actually been detected and confirmed as correct. That is, RXDM first attempts to correlate any SAP state-change event that it detects with any pending directives to which it might correspond before it finally determines to transmit an informatory event message. For instance, after an explicit \texttt{STOP} command has been issued by the user to a particular SAP, the monitor enters a mode in which it awaits an expected state-change event for that SAP from 'running' to 'stopped'. The detection of this event is considered part of the normal execution sequence for the \texttt{STOP} command. When the 'stopped' event finally is detected, the monitor does \textit{not} transmit an event message packet to the user; rather, in this case the stop is simply considered confirmed, and the monitor completes execution of the \texttt{STOP} command sequence by transmitting a \textit{reply} message packet back to the user, corresponding to the original \texttt{STOP} command message, which indicates its successful execution.

On the other hand, it is possible that the detected state change does not correspond to what is expected. In the above \texttt{STOP} example, for instance, the targeted SAP may actually terminate for some reason, or simply exit, rather than enter a 'stopped' state as directed. Note that this is feasible since the operating system under which the monitor and the SAP's are running may not be 'real-time' (certainly UNIX is \textit{not}); \textit{e.g.}, there is no guarantee inherent in any given UNIX implementation that a process won't have a chance to do something on its own between the time another process sends it a signal (\_\texttt{SIG\_STOP} or \texttt{SIGKILL}) and the time the signal is actually delivered and enforced by the operating system. (Additionally, there may already have been a pending signal sent from a third party process that will be delivered before the current signal.) In this situation, RXDM will first transmit to the user an informatory event message packet relating the detection of an \textit{unexpected} change of state (\textit{e.g.}, 'termination') of the target SAP, before finally transmitting an \textit{error}-reply message packet corresponding to the pending \texttt{STOP} command, which is
therefore interpreted as having *failed*.

It is worthwhile to note, however, that the state-change detection scheme employed in RXDM is not so discriminating as to detect and ensure that the *cause* for an expected SAP state-change is in fact the successful implementation of a given directive. (Perhaps it is even questionable whether or not it matters that one knows the answer for certain!) That is, it is possible for the expected change of state to occur, but as a result of a cause unrelated to the directive whose execution is pending. In general, although these situations are often detectable, RXDM is designed not to care *how* the expected state change happened, only that it actually *did*. In these cases RXDM interprets the directive as having *successfully executed*, and it does *not* issue an informatory event message previous to the reply message for the directive, as it did in the error-case described in the above paragraph. A hypothetical situation manifesting this phenomenon would be, say, when a *TERMINATE* command is implemented for a particular SAP *via* issuance by the monitor of a UNIX *SIGHILL* signal, but the SAP actually terminates due to an internally-generated signal such as an untrapped *SIGALRM*.

The next situation in which the monitor will generate an asynchronous event message packet is when it detects data is available on one of the *indirect communications links* corresponding to an SAP output channel (as opposed to a *direct* communications link between the SAP and the user, external to the monitor). That is, normally, when an SAP is first started, RXDM sets up a connection between itself and the SAP’s standard input and output channels *via* the usual redirection facility for communications. Under UNIX, an SAP is in fact a child process of the monitor, so this amounts to redirecting the child process’ file descriptors 0 (SAP standard input channel), 1 (SAP standard output channel), and 2 (SAP standard error output channel) through a specialized, 2-way ‘indirect’ communications link (UNIX pseudo-terminal, or *pty*) previously created by the monitor (*i.e.*, the parent process). This redirection is done right after the *fork()* and before the *execv()* for the SAP. Actually, it is in fact possible to establish similar indirect communications links to any other extant SAP input/output channels in addition to these standard channels, as indicated by the user, at the time the SAP is started up — all such links corresponding to SAP output channels are thereafter continually monitored for the presence of output data. The specification of which SAP input and output channels are to be connected as indirect *vs.* direct (or not at all) is communicated by the user within three separate bit-vector data fields in the *START_PROG_*x command message packet for the SAP. Note
that the standard I/O channels referred to above are connected as indirect communication links by default, without need of explicit specification via these data fields (although this default action may be overridden by a special command option).

After an SAP is started, its indirectly-connected output channels are regularly checked by the monitor for the presence of output data. This happens as frequently as is practical in order to achieve a prompt response in forwarding SAP output data to the user without unduly burdening the monitor. Normally, any data detected is read in by the monitor as a simple, unstructured stream of bytes, and is then immediately transmitted to the user, without transformation, via one or more program output data event message packets.

In the case of an SAP generating large amounts of data in a short period of time, however, this simple scheme of 'detect, read, and send' might result in 'swamping' of the user and/or the monitor due to the amount of program output data being processed (or rather, the number and rate of data event message packets being issued as a group). For example, the monitor could become substantially I/O bound, so tied up just handling the massive data output stream from a pathological SAP that it could no longer carry out its other responsibilities in an effective and responsive way. Furthermore, even though the user might expect to handle such an SAP if he were directly connected to it (and not indirectly via the monitor), the user now has the added burden of the communications overhead associated with dealing with the data event message packets themselves, so it's possible the user could also become swamped in this situation! Consequently, RXDM is designed to limit the maximum number of such packets sent as a group each time it checks for output data from a given SAP, which effectively clamps the rate at which SAP output data is processed.

Note that besides this normal method of dealing with SAP output data, RXDM provides other options to the user as well. One is that the user can specify (via the REDIRECT_PROC_IO command) that the data be forwarded to another destination that is local to the monitor, such as a file, or even the input channel of another SAP, rather than to the user via a data event message packet. This is, of course, similar to the typical output redirection mechanism provided in UNIX command interpreter shells (the corresponding input redirection mechanism is also provided by RXDM). Another option is that the user can specify that the usual data scan carried out by the monitor be completely disabled for a given channel for a given SAP (and then reenabled at a later time, as required). For some applications programs, this would probably
result in the program becoming blocked from running after the system communications link buffers were filled without being drained by the monitor.

Another situation in which RXDM issues an asynchronous event message is when the monitor detects that it has committed an *internal error*. An internal error basically consists of any situation that leads to an inconsistent and/or irreconcilable state of the monitor, and could be caused by anything from an unanticipated system call failure to a bug in the software. The transmitted error event message packet contains error code fields indicating the general nature of the problem, a code indicating the location in the software where the error was detected, the ascertained code for the pending user directive (if any) and its associated message sequence number, plus additional fields for any peripheral information relevant to interpretation of the internal error.

The potential internal error for which the monitor is most heavily instrumented is the occurrence of an irreconcilable communications failure during the attempt to transmit a *reply* message packet (of any type) to the user. Of course, it is possible that the subsequent attempt to send the error event message packet could likely fail as well, except for the fact that there are differences in the approaches used in transmitting asynchronous event messages versus reply messages. Namely, the monitor may be configured so that the two message types travel across separate communications links (e.g., individual UNIX sockets) between the monitor and the user (this is established when the monitor is first started by the user). Furthermore, the monitor's communications system is a multilayered hierarchy of mechanisms involving queueing of incoming and outgoing messages. The attempted transmission of the reply message packet could have failed within some monitor communications layer preceding that which flushes the monitor's outgoing message queue buffers. The error event message (like most asynchronous event messages), on the other hand, always forces a flush of this queue first, and then is itself sent directly via the monitor's lowest-level communications layer mechanism (byte pump), without being enqueued.

Finally, the RXDM monitor will generate asynchronous event messages associated with the intermonitor communications connections (if any) between itself and other monitors on the same machine. This special intermonitor communications (`talk`) facility will be discussed in greater detail in section 4.3.3. In particular, the monitor will send an event message to the user whenever another monitor is attempting to initiate a new intermonitor connection. Also, once an intermonitor connection exists,
the user may optionally configure it (using the CONTROL_CHANNEL monitor service request) to be sensitized for the arrival of incoming data from the other monitor, such that this data is automatically and asynchronously forwarded on to the user in the form of an event message packet (similar to the way data detected on an indirect output channel for an SAP is handled).

4.2.4 Data Compatibility: Representation of Typed Data Items in RXDM Messages

Most message packets communicated between the user and the monitor contain typed data items. The most common data types communicated are integers and SAP address values. As previously mentioned, the RXDM monitor is designed in accordance with the model delineated in Chapter 3. That is, it is designed for operation in a conglomerate (i.e., inhomogeneous, distributed) computing environment. This poses the problem of compatibility conflicts in the representations of typed data items that are communicated between the individual, potentially incompatible computational elements that constitute the conglomerate (esp. viz. machine architectures under which the user and monitor are running). For example, the two architectures involved may differ in terms of byte order, word (integer) length, and even the format of some of the simple primitive data types (e.g., integer sign representation, floating point mantissa and/or exponent representation). (Note that 'simple primitive data types' here refers to the following data types that are assumed to always map directly onto hardware: byte/char, integer, floating point, pointer/address, and word (unsigned integer).) Compound or multiple-precision primitive data types (i.e., those primitive data types whose implementation is on some systems software (i.e., compiler)-dependent, such as short/long integer or double precision floating point) can also present an incompatibility problem due to differing precisions/lengths. The macroscopic computational issues that can arise in a conglomerate computing environment as a result of these various data incompatibilities has been previously discussed in the Chapter 3.

To help overcome this problem, RXDM obtains specific knowledge about the user's computing environment, and conversely provides the user with specific knowledge about its own computing environment. This is accomplished as part of the startup/initiation protocol for the monitor, in which an exchange is made of fundamental information about each of the two systems, via the XCHG_ENVIR_INFO com-
mand. This information includes: codes indicating the byte order and integer sign representation (even though two’s complement is predominant in almost all modern architectures); floating point representation characteristics (single and double precision; note the IEEE standard is not assumed); machine word size; number of machine registers; the indices for special-purpose registers such as the program counter, program status word, stack pointer, and stack frame pointer; the size of the primitive data types (integer, long integer, floating point, pointers, etc. — note that the size of pointers/addresses on some architectures (especially for those with a segmented address space) does not exactly equal that for an integer, and conceivably could even differ from the machine word size!); and, codes for the operating system (e.g., UNIX) as well as the machine architecture itself. Of course, one could simply send the latter code, indicating the particular architecture, in lieu of all the other codes relating specific characteristics; it would then be the responsibility of the receiver of this code to look up the individual architectural characteristics, as needed, from a table. This is not desirable, since the possible architectures with which communications can take place is therefore restricted by the predefined database of lookup tables. In contrast to this, with the scheme used in RXDM whereby all the relevant characteristics are communicated in the environment information exchange process, the case in which the code for the machine architecture is not recognizable should not affect the communications capability in any negative way. In fact, the user's machine architecture code is completely unused by the monitor; the monitor's architecture code is nevertheless sent to the user for informative purposes.

Note that the information exchange itself clearly involves message passing, but the data transformations are obviously unavailable at this point. The resulting circular conflict is avoided by defining the type of each data item to be a single-byte entity, since the size of a byte is assumed to be invariant (8 bits), and the relative bit ordering within a given byte is assumed to be an invariant for each individual computational micro-environment (network plus individual architecture). This condition of course restricts the value range of each datum to that containable in a single byte (0 to 255, or -128 to +127), which nevertheless turns out to be realistic since most of the required information is simply small-valued codes or flags.

Once the environment information is available, typed data items (i.e., data parameter fields) in all subsequent message packets undergo a data normalization process, in which the item representations are transformed to a ‘normalized’ form by the sender of the message packet for communication across the network, and likewise
transformed to the 'denormalized' form (most likely, the usual representation in the
local computing environment) by the receiver of the message packet. Despite the im-
lications inherent in the word 'normalized', however, the precision of the normal-
ized, transmitted data item always is that corresponding to the sender's environ-
ment. This method was chosen so that there is no fundamental restriction on the pre-
cision of transmitted data, so that the receiver of the data is given the option as to
how much of the transmitted precision is to be retained vs. truncated (i.e., all the
data is there if he can use it — consider in particular the case of floating point data!).
Note that the data items that are pointers/addresses in the address space of the
sender pose a special situation, since these usually must be retained and manipulated
by the receiver according to their actual precision. Therefore it is convenient to view
such items as abstract objects (actually, this example serves as a harbinger of a more
general treatment of remotely-generated data items as abstract objects, to be dis-
cussed in Chapter 5).

4.2.5 File System Interface and Special Monitor File Abstractions

The monitor provides the user with a variety of operations for files located on the
monitor's local file system. These are the usual open, read, write, close, and posi-
tion (for random access) operations. The corresponding RXDM commands are
OPEN_FILE, READ_FILE, WRITE_FILE, and CLOSE_FILE (the position operation is dis-
cussed as follows). However, these operations are not just simply designed to fill in
for situations in which a globalizing network file system (e.g., NFS) is unavailable,
but also to enhance the functionality of the RXDM monitor's operation in a conglomera-
tive computing environment by means of the extension of their applicability
to specially-defined kinds of file abstractions.

The READ_FILE and WRITE_FILE commands can take a random file position or
address as an argument, and therewith effect a file positioning operation previous to
starting the data transfer operation. Furthermore, the usual options are available for
these commands that specify that the given file position/address argument is to be
interpreted relative to one of the following: the current file position; the beginning of
the file; the end of the file (in which case the given position/address argument is im-
plicitly a negative value); or, as an absolute file address (for disk files, this is the
same as 'relative to the beginning of the file'; however, this is not the case for some
other file types implemented by the monitor, as discussed below).
A file may be positioned without any data being transferred (i.e., implementation of the file position operation) by simply specifying a zero-length data transfer argument in either of these commands. Also, since these commands always return the (new) file position after the data transfer operation has been completed, then it is possible to ascertain the current file position nondestructively by simply specifying a zero-length data transfer argument plus a zero file position argument relative to the current file position.

The monitor provides several special features that extend the usual operations enumerated above, both in regards to the semantics of the data transfer involved (i.e., transfers of typed data items, vs. simple bytes, may be specified), as well as kinds of files accessible (normal disk files vs. special, new abstract file types).

The first such feature permits the read, write, and position operations to reference typed data items, instead of just the usual byte datum. The allowed data types are the primitive types discussed previously in section 4.2.4 (e.g., integer, floating point, etc.). This feature is provided as a convenience to the user, in accordance with the design goal of overcoming the potential problem of data incompatibilities between the user's and monitor's computing environments. By abstracting data transfers to reference typed data items, and consequently hiding the machinery involved in conversions between a contiguous block of typed items and the corresponding contiguous block of bytes, there is no explicit need for the user to do special manipulations of the read or written data items in case the environments (i.e., data representations) do indeed differ. This feature may in fact be viewed as the natural extension of the file concept in UNIX, from files as arrays of bytes, to files as arrays of typed data objects, thereby giving files that fundamental aspect of structure necessary to facilitate efficient and meaningful exchange of data in a conglomerate computing environment. More concisely, the invariant data units in the distributed computational environment are extended from bytes to the basic primitive types, such as integer and floating point values.

Yet another capability provided by this feature is the possibility of data type overlays involving even more complex data structures. For example, the user can efficiently reference a member of a structure in an array of structures stored on a file based on an offset computed in integer equivalents (say), instead of byte equivalents. Again, this is possible since the integer data type is abstracted to be a legitimate invariant in the computational environment by the (low-level) mechanism of data nor-
malization discussed above.

The second major feature provided for file manipulation operations is that the kinds of files accessible is extended to include a special set of abstract files, including the following:

- abstract files defined to correspond to the indirect SAP input/output channels (as discussed above), in order to permit the user to have communication with such channels by means of the usual read/write file operations
- regions (contiguous blocks of memory allocated within the monitor's own address space)
- program environments (special regions that exclusively contain environment bindings for application by the monitor to SAP's)
- the special intermonitor call file abstraction (for initiating/establishing a communications path between two individual monitors running on the same machine but with possibly different users, as a means of implementing a 'talk' facility between the users)
- and, perhaps most significantly of all, the abstraction of the text and data spaces of each SAP as contiguous files.

The latter is a very powerful abstraction. One can now view the text space of an SAP (obtained via an OPEN_FILE command with the special file type OPEN_FILEcmd_PROGTEXT) as an array of instructions (an 'instruction' is implemented as a primitive data type), which permits convenience and flexibility in examination and modification of program text (e.g., setting of breakpoints, or the patching-in of special text segments specifically intended for facilitating the debugging process). Likewise, one can view the data space of an SAP (obtained via an OPEN_FILE command with the special file type OPEN_FILEcmd_PROGDATA) as an array of typed data items. In both cases, this abstraction very significantly uncouples the user from the burden of having to view program addresses as architecturally-dependent, abstract data objects, and permits him to view them as simply integer-valued, positional offsets relative to the start of a file (or, as indices into arrays of the appropriate data items). Since offsets are simply integer (rather, as used in the RXDM monitor, long integer) quantities, the user may consequently feel free in performing address arithmetic based on these offsets (assuming that he has sufficient integer precision available to do so), which he couldn't easily do using the architecturally-dependent program ad-
dress objects! Figure 4.4 illustrates the mapping of abstract SAP text and data space files to the physical address space of an SAP.

![Figure 4.4: SAP Address Space and Abstract Text and Data Files](image)

Especially relevant in this regard is the application of such abstractions to the runtime program stack. Although this stack is dynamic, there is the potential on some architectures to abstract the stack (at any given time) as consisting of a fixed array of frames, whereby each frame of course corresponds to a particular instance of a previously-called procedure (context history) in the SAP, and contains the usual in-
formation such as saved register contents (including the pending value for the program counter), procedure parameter values, and (possibly) storage for local variables.

One additional special feature regarding extended file abstractions is that a file may be opened to overlay another, extant file, with the option of an offset from the beginning of the old file defining the start of the new file, as well as a length for the new file that may be less than that of the older. The new file in this case subsumes all other characteristics of the old file, including the file type, but may begin and end at arbitrarily-selected positions relative to the old file. This overlay operation is performed by first positioning the extant file to the desired location that is to correspond to the start of the new file, then performing the OPEN_FILE command with the special file type OPEN_FILEcmd.OVERLAY specified in order to actually create the new overlay file. The old file may then be repositioned to its original location. The ending location of the new file will be the start location.

This notion of file overlays could be especially useful as applied to the SAP text and data space file abstractions. For example, the user may elect to open a separate file corresponding to the text of each procedure in an SAP, whereby each such file is defined as an overlay onto the original text space file, with the appropriate starting offset and length. Or, the user may open a separate file for each of several data structures of interest in the SAP data space (e.g., one for each separate frame on the stack, up to the obvious limit on available files, or, a separate file for registers and for data parameters in a certain (fixed) frame). This is a very convenient feature for separately accessing and modifying elements of two or more different arrays in an SAP's data space (either static or dynamic data), especially in combination with the aforementioned ability to reference the arrays on the basis of typed data elements vs. bytes. (However, as with any such mechanism providing multiple access pathways into a data set, the user should of course exercise the appropriate caution regarding avoiding a situation in which an unexpected absence or presence of modified data results, as a consequence of a naive and/or careless sequence of operations on equivalenced areas of physical storage!)

All file operations (except open) take as their target a small integer representing the actual file, similar to the 'file descriptor' in UNIX, called the fileindex. This is returned to the user by the OPEN_FILE command upon successful completion of the open operation. The monitor maintains for each open fileindex a 'file control block' providing the usual information required for file access, such as pointers indi-
cating the start, ending, and current file positions. The monitor consistently attempts to perform some checks on the the user's file operations before actually proceeding with their execution, including the obvious checks for ‘file open’ and out-of-bounds accesses. The current limit imposed by the monitor on the number of open abstract files of any type (besides any system-imposed limits) is 64.

4.2.6 Controlling the Execution State of Supervised Applications Programs

In debugging an applications program, it is obviously very important for the user to be able to control its execution manually; for example, asynchronously stopping the program in order to examine the contents of its registers or set/reset a breakpoint, and then continuing with the execution of the program. These control operations also have utility in other areas as well, such as process coordination and synchronization. It is therefore probably useful to distinguish between separate notions of program state in relation to the control of a program’s execution, versus the instantaneous description of its ‘data image’ in memory. Consequently, I define the generalized notion of the execution state of an applications program, which consists of its essential ‘running’/‘not running’ status, plus the contents of its instruction or program counter (which, from one system to another, is almost without exuception a user-accessable machine register (the PC)), at any given point in its execution. In the following section I likewise define the notion of the associated data state of a program. This section, then, is concerned with the description of the operations provided by RXDM for the examination and modification of the execution state of the SAP’s under its care.

The RXDM monitor in fact provides the expected set of operations for the control of execution of its SAP’s. These include the start, stop/pause, continue, single—step, and terminate operations for modifying an SAP’s ‘running’/‘not running’ status. These operations effectively interact with the system, communicating to it the desired run-status of an SAP process that it is to effect (note that, under UNIX, an SAP is always a child process of the monitor process). In addition, the stop, continue, and single-step operations are also designed to provide for the examination and random modification of the SAP’s program counter (PC).
The start operation may take one of three different forms (ref. the monitor commands `START_PROG_TRACED`, `START_PROG_ATTACHED`, and `START_PROG_NOTRACED`). The forms correspond to different modes of interaction between the RXDM monitor and the started SAP.

In particular, the first form (ref. `START_PROG_TRACED`) implements by way of special, debugging-oriented system calls (e.g., the `ptrace()` system call in UNIX) the ability for the user (monitor) to examine and modify the registers and text and data spaces (plus, on some systems, the so-called process space — see below) of a traced SAP. Also, a traced SAP displays the special behavior that, whenever it is about to change state from 'running' to 'not running', it is simply 'stopped/paused' instead. This enables the monitor to capture a state change before it is actually effected, and then report the pending event back to the user in the form of an asynchronous event message packet. Consequently, the user then has the opportunity to direct the monitor to actually effect the pending state change in the SAP process, or to modify the process (or even the nature of the pending state change) and then continue its execution.

One special note here is that, a newly-started traced SAP always is left in a special 'stopped/paused' state immediately after having its execution initiated using this form of the start operation, regardless of whether the `START_PROG_TRACED` command was issued as blocking or nonblocking.²

Also, with this form of the start operation, the user is provided the ability to single-step a non-running, traced SAP (at the machine instruction level), using the

² That is, this is the normal state of a traced SAP process immediately after successful completion of the start operation by the monitor — the SAP has effectively not executed any instructions in its text space, behaving in a manner analogous to its having encountered a 'breakpoint' immediately before execution began (except that this special event is captured and absorbed by the monitor as a normal part of the startup sequence, with the result that the monitor does not issue an asynchronous event message packet to the user in this case, as it would for a 'normal' breakpoint). This feature provides the user with an initial opportunity to make whatever modifications or examinations of the SAP registers and/or text and data spaces that he desires, prior to the actual start of execution of the SAP. For example, he may wish to initialize certain of the SAP's data structures, or set breakpoints. The user must subsequently issue a separate `continue` operation in order to specifically cause the traced SAP to actually begin its execution.
monitor SINGLSTEP command as described below.

The second form of the start operation (ref. START_PROG_ATTACHED) is only available on certain systems (in particular, on the Sun Microsystems line of computers running Sun UNIX), and is similar in nature to the first form of start operation described above. The major difference is that the operation does not return with the newly-started, attached, traced SAP in an initial, 'stopped/paused' state; rather, the SAP immediately starts executing in the normal manner as soon as it is invoked by the monitor. (Note that the user can, of course, subsequently force the attached SAP into the 'stopped/paused' state explicitly via a STOP command.)

Basically, an attached, traced SAP started with this form of the start operation displays the same characteristics and behavior as a traced SAP started with first form described above. One additional difference, however, is that the user may, if desired, remove the tracing feature altogether, for whatever reason (efficiency?). This may be accomplished via the monitor DETACH_TRACE command (and likewise reinstated again via the ATTACH_TRACE command).

The third form of the start operation (ref. START_PROG_NOTRACED) simply provides for initiation of the 'normal' mode of execution for a nontraced SAP. The user cannot examine or modify the registers or text and data spaces of a nontraced SAP (although he may transform it into an attached, traced SAP via the aforementioned ATTACH_TRACE command, and then perform such operations!). This means that breakpoints cannot be set for a nontraced SAP as well. Furthermore, unlike a traced SAP, when a nontraced SAP changes state from 'running' to 'not running', the monitor does not have the ability to capture the state change before it has actually been effected. Instead, the monitor can only detect the change 'after the fact', and then report this event back to the user via the usual asynchronous event message. Thus, the user has little opportunity (or none, in case the SAP process actually terminated execution!) to examine/modify the course of execution taken by such an SAP, other than through the usual STOP, CONTINUE, and TERMINATE commands. Note that single-stepping is not possible for nontraced SAP's.

Finally, note that each form of the start command takes the same set of arguments. These include specification of the pathname (file name and optional directory name) for the executable file for the program on the local file system, and a list of 'command line' argument strings (ref. argv[] / argc in C) to be made available to the newly-started SAP. In addition, the user may optionally specify a limit on the to-
tal running time taken by the SAP, as well as the direct and indirect data communications links to be created for the SAP (see the corresponding discussion in section 4.2.3).

Each time an SAP is started, the monitor will simulate an internal 'change directory' operation to the directory whose name is specified as part of the path name for the executable file. This *current working directory* remains in effect until again changed by either another start operation, or explicitly via a CHG_KERNWDF command. The initial value for the monitor's current working directory (cwd) is the root directory for RXDM. Note that, as an option for the start operation, the user may specify the special directory name '*' to indicate that the presently active cwd is to remain active across the start, or the special name '~' to indicate a change of the cwd back to the original RXDM root directory. Furthermore, any file name specified in an OPEN_FILE command is assumed given with respect to the monitor's cwd, so a complete path name is not necessary if the cwd is appropriate. Alternatively, the file to be opened using OPEN_FILE may be explicitly specified using its complete path name; this does not affect the monitor's cwd, which, again, may only be altered using a start operation or the CHG_KERNWDF command.

With the successful completion of all these forms of the start command, the monitor returns a small integer to the user, which serves to identify the SAP in all further references made to it by the user (e.g., as in the STOP command), called the progin dex. This approach was taken for the obvious reason that using the complete file path name (or some other name string) to identify the SAP in subsequent commands targeted at that particular SAP would be extremely inefficient, indeed. The 'current working directory' for the monitor will also have been changed to that specified in the start command. This feature permits the user to specify "." for the directory part of the path name for subsequent start operations, thereby indicating that the executable file name of an SAP is to be found in the current working directory. Again, this approach was chosen in order to help improve overall communications efficiency by shortening the message packet lengths, at least for the potential situation in which there is a large and frequent number of SAP start operations happening. Note that, under certain circumstances, the monitor's current working directory will be set to the value specified in the start command even if the start operation for an SAP fails. Consequently, the user should always explicitly re-specify the directory name part of the path name of the executable for the next start operation attempted, after a start operation that has failed.
As mentioned, the RXDM monitor also provides operations for the control of execution of an SAP after it has been started; namely, *stop* (ref. the *STOP* command), *continue* (*CONTINUE*), *terminate* (*TERMINATE*), and *single-step* (*SINGLSTP*). The description of these commands is fairly obvious, with some minor comments, as follows.

The *STOP* command forces a running SAP into the 'stopped/paused' state. Note that this command has no effect on an already-stopped SAP.

The *CONTINUE* command forces a non-running SAP into the 'running' state. It has no effect on an already-running SAP, nor can it be used to force an already-terminated, nontraced SAP to run (i.e., such an SAP process has dissapeared). The user may optionally specify in this command that the SAP is to resume execution at a specified address, as well as an optional signal (ref. UNIX *signals*) to be delivered to the SAP at the time execution is resumed.

The *TERMINATE* command forces an SAP, running or not, to terminate execution. It is not possible to resume execution *via* the *CONTINUE* command of a terminated SAP process.

The *SINGLSTP* command causes a 'stopped/paused', traced (only) SAP to execute the (single) machine instruction located at the address currently contained in the PC register, and then reenter the 'stopped/paused' state. The user may optionally *randomly* specify the address of the instruction to be executed. Note that this operation is normally utilized to 'step across' a breakpoint that has caused the SAP to stop running. This procedure involves the following sequence of operations:

- the traced SAP encounters a breakpoint and enters the 'stopped/paused' state
- the monitor detects the SAP state change due to the breakpoint and issues a corresponding asynchronous event message to the user, containing the address in the SAP where the breakpoint occurred
- the user temporarily reinstates the original instruction at the given address, using a *poke* operation, as described below
- the user issues a *SINGLSTP* command to the monitor to force execution of the reinstated instruction (only)
- the user again installs the breakpoint at the given address, using a *poke* operation
the user issues a CONTINUE command to the monitor in order to force
the SAP to resume normal execution at the address immediately follow-
ing the breakpoint location.

Also note that, when the STOP, TERMINATE, or SINGLSTP commands are applied
to an SAP (either traced or nontraced, blocking or nonblocking), then the monitor
normally effects the respective state-change action without the subsequent transmis-
sion to the user of an asynchronous event message, which it would otherwise do in
case an SAP changed state unexpectedly. However, if the monitor detects that the
targeted SAP doesn't respond to these operations in the expected manner, then it will
in fact in certain situations (as deemed appropriate) issue an asynchronous event
message to the user — refer to the discussion of this topic in section 4.2.3.

4.2.7 Controlling the Data State of Supervised Applications
Programs

An essential part of any debugger is that it provide a means of accessing and modifying
the contents of specified locations in an applications program. This applies equally
well to the program's text, or instructions, and the program's data areas (registers,
static data, and dynamic data, including the heap and the stack). Consequently, I
will generalize that part of a program's state that is not directly concerned with its
execution status to encompass both of these notions — namely, that an applications
program's data state is defined to be the instantaneous description consisting of the
entire contents of its registers plus its text and data spaces at any given point in its
execution. (Notice that the definitions of execution state and data state therefore
only intersect as a result of the inclusion in both concepts of the special-purpose pro-
gram counter (PC) register (i.e., its contents).)

The RXDM monitor provides a set of features aimed specifically at facilitating in-
spection and modification by the user of the data state of an SAP. (Note that all such
features apply only to SAP's that were started with tracing enabled using either the
START_PROG_TRACED or START_PROG_ATTACHED command.) These include the usual
operations for examining the contents of a particular location in an SAP's text or data
spaces (peek), as well as the modification thereof (poke). However, these operations,
like the RXDM file system operations discussed above, have been extended with RXDM
to operate on typed data objects (i.e., viz. the primitive data types discussed in sec-
tion 4.2.4), yielding, for example, commands such as `PEEK_DATA,DBL` (examine double precision floating point-valued contents at the specified location in the SAP data space) and `POKE_TEXT_INSTR` (modify instruction at the specified location in the SAP text space).

The monitor also provides specialized peek and poke operations for an SAP's *process space* — that is, insofar as permitted by the system. These operations are `PEEK_USER` and `POKE_USER`, which allow examination and modification of the contents of specified locations in the process space maintained by the system for each active SAP. This information is obviously very architecture- and system-dependent. Consequently, as data items, the contents of such locations are assigned the primitive type 'kernword' by the RXDM communications mechanism, and as a result are treated as abstract data objects that do not undergo any form of data normalization transformation during the message packet communications process, as do, for example, integers. Note that these operations are very low-level indeed, and require special knowledge on the part of the user as to the layout of the information stored by the system in an SAP's process space!

The monitor also provides special commands for manipulation of an SAP's machine (rather, process) registers, including the program counter. These operations include `READ_REGS`, which returns to the user a list of the contents of all (non-floating point) registers defined on the host machine. There is also the corresponding `WRITE_REGS` command, which sets all the registers according to a list of values supplied by the user. Note that the number of registers is architecture-dependent. Consequently, these lists can have different lengths according to the particular architecture on which RXDM (and the SAP's) reside; the actual number of registers involved is thus also transmitted (actually, this parameter is embedded within the list as part of the so-called 'list header', in accordance with the standard construction in RXDM of lists of typed data items — see the discussion in the following section). As data items, the register contents are assigned the primitive type 'kernword' by the RXDM communications mechanism, and as a result are treated as abstract data objects that do not undergo any form of data normalization transformation during the message packet communications process. The user should deal with them accordingly.

The designation of which registers in the register set are special-purpose (*e.g.*, the program counter) is also architecture-dependent. However, recall that the user already has this knowledge supplied to him as a result of the initial environment infor-
mation exchange (ref. the \texttt{XCHG\_ENVIR\_INFO} command) performed during the startup/initialization phase of the monitor (see section 4.2.4). Namely, at that time the indices for the following special-purpose registers were supplied: the program counter (PC); the program/process status word (PS); the stack pointer (SP); and, the frame pointer (FP).

In addition to the \texttt{READ\_REGS} and \texttt{WRITE\_REGS} commands, which operate on lists of values for the entire register set, the corresponding commands \texttt{GET\_REG} and \texttt{SET\_REG} are provided, which take a single register as their argument. The \texttt{SET\_REG} command also returns the original contents of the modified register.

The user may utilize the peek and poke operations to perform the remote setting/deleting of breakpoints in an SAP's text space, assuming that the user is knowledgeable as to the appropriate value to use for the breakpoint instruction on the given architecture. Consequently, RXDM provides, as part of the aforementioned environment information exchange, an appropriate instruction value to be used for the breakpoint (normally either an illegal instruction, or a special-purpose trap instruction explicitly dedicated for use in debugging on the machine on which RXDM resides). In conjunction with this, another special instruction value is supplied by RXDM that is orthogonal to all valid machine instructions. This value may be utilized, for example, within the user's 'breakpoints database' (which the user maintains in order to keep track of such information as the location, original (pre-breakpoint) instruction value, and active/inactive status of all breakpoints in each (remote) SAP) in order to indicate that the breakpoint is \textit{defined}, but \textit{uninstalled} in the sense that the user has not yet physically replaced the original instruction at the target location with the breakpoint instruction. Note that these two values are assigned the primitive type 'kerninstr' by the RXDM communications mechanism, and as a result do not undergo any data normalization transformations during transmission by the monitor, and should be treated by the user as purely abstract data objects.

When a user-installed breakpoint is encountered by an SAP during its execution, the SAP will of course change state from 'running' to 'stopped/paused'. This state change, as previously discussed, will automatically be detected by the monitor, which then immediately reports the state change event to the user in the form of an asynchronous event message packet. The monitor will also include in this message the location (address) in the SAP's text space where the exception occurred, as well as the contents thereof (namely, the breakpoint instruction value), and a code indicating
that the exception *probably* corresponds to a breakpoint (as opposed to just 'some illegal instruction') in the SAP. Note that the term 'probably' is used here since, insofar as the monitor is aware, it is possible that the presence of the offending 'breakpoint' instruction may be the result of some source besides the user, who himself may not have actually installed a breakpoint at that location via a peek operation! Consequently, the user normally should consult his 'breakpoints database' in order to verify that a breakpoint actually was installed at the location reported in the event message..

4.3 Special Features

In this section, I discuss enhancements to the basic design of the monitor that are intended to provide the user with extended capabilities regarding many of the tasks indigenous to a large distributed conglomerate computing environment. Consequently, many of these enhancements have as their goal the remediation of the user's having to deal with architecture-dependence in this type of computing environment. The primary method used in achieving this goal is to base the enhancements on the SAP text and data space file abstractions discussed in proceeding sections.

4.3.1 Local Breakpoints Facility

As an alternative to this 'remote' method for the setting and deleting of breakpoints described previously in section 4.2.7, which is based upon the use of low-level peek and poke operations by the user, the RXDM monitor provides a complete 'local' facility for the manipulation of breakpoints. This facility is specifically designed to provide the user with a *completely architecturally-independent abstraction mechanism for the usual breakpoint manipulation operations*, and comprises the commands SET_BRKPT, DELETE_BRKPT, CONT_FROM_BRKPT, and CLEAR_BRKPTS.

All these commands operate on abstract text files corresponding to SAP program text space (i.e., files created with the special type \texttt{OPEN:\_FILE\_cmd\_PROGTEXT}, or overlays thereon, as described previously in section 4.2.5), taking a \texttt{fileindex} as an argument. The SAP process associated with the targeted abstract text file must be in a 'not-running' state. Naturally, the \texttt{SET_BRKPT} command also takes as an argument the
location of the breakpoint, which may be specified as an integer offset defined relative to either the start of the text fragment to which the abstract file corresponds, or relative to the file's current file pointer value (as set by either a previous READ_FILE or WRITE_FILE command). Alternatively, the location of the breakpoint may be specified as an absolute address in the program text space, not defined relative to the start location or the current file pointer value of the corresponding abstract file. However, in this case, the restriction still applies that the specified address must fall within the existing physical start and end address boundaries defined for the given abstract text file (which may be logically different than the start and end addresses for the original program text space, since the file may have been created as an overlay — for example corresponding to a particular subroutine in the original program).

The SET_BRKPT command also permits the user to define more than one type of breakpoint at the same location. However, the actual interpretation of the 'type' is left up to the user; that is, insofar as the monitor is concerned, the type of the breakpoint is simply a distinguished data constant. (Note one restriction: the monitor expects that all breakpoint types must be distinct from each other.)

Using SET_BRKPT, the definition of breakpoints in an SAP's text space may be accomplished via more than one abstract text file, since multiple text files opened for the same SAP provide multiple paths into the SAP's text space. However, attempts to multiply-define a breakpoint at the same physical address in the SAP and with the same type are ignored by the monitor, whether the attempt is made through a single, or multiple abstract text files.

In accordance with the RXDM design philosophy for referencing SAP's abstract files via small integer indices, the SET_BRKPT command likewise returns to the user an integer for referencing the newly-created breakpoint object, known as the bpindex. (Actually, the user must subsequently identify a particular breakpoint using the pair <fileindex, bpindex>.) Once again, this is done for the sake of efficiency in communications. The DELETE_BRKPT and CONT_FROM_BRKPT commands consequently take the fileindex and bpindex as arguments.

The CLEAR_BRKPTS command is used to delete all extant breakpoints of a given type that were defined through previous applications of the SET_BRKPT command on the specified abstract text file. It takes as arguments the respective fileindex and breakpoint type. Also, note that a CLOSE_FILE operation on the abstract text file will implicitly perform a CLEAR_BRKPTS on all breakpoints of any type defined on that
Note that since the definition of a breakpoint of a given type is associated with a unique abstract text file, then its deletion using either the DELETE_BRKPT or CLEAR_BRKPTS command operating on the corresponding fileindex is complete, in the sense that there will be no other multiple definitions in existence to deal with. Also, if more than one breakpoint type is defined (possibly through various abstract text files) at the same physical location, then it is necessary to delete all the various types of breakpoints at that location before the monitor actually removes the physical breakpoint implemented at that location.

The CONT_FROM_BRKPT is a special command that may be used to force an SAP to 'step across' a local breakpoint that had earlier caused it to enter a 'stopped/paused' state, thereby continuing with its execution (in effect, the monitor implements the 'step across' sequence as discussed above for remote, user-implemented breakpoints). This command takes as arguments the fileindex and bpindex identifying a local breakpoint from which execution shall continue (not necessarily the same breakpoint that caused the stop!). Note that when the monitor detects that an SAP has stopped running due to encountering a local breakpoint, then it notifies the user of this fact (as distinguished from the case that remote breakpoint is encountered), specially providing a fileindex and bpindex identifying the breakpoint, by means of an asynchronous event message corresponding to the SAP's run-state change (see section 4.2.3).³

Finally, one other characteristic of local monitor breakpoints is that they are hidden by the monitor. That is, the user may never 'see' them by using any method of inspecting the contents of the SAP text space, whether through a low-level peek operation or through a READ_FILE command directed at an abstract text file defined on the SAP text space! Instead, the monitor ensures that the user will always see the original contents of any text location at which a local breakpoint is defined.

³ Note the choice here of the phrase 'a fileindex and bpindex' to indicate that, if the local breakpoint is multiply-defined in the sense of more than one type defined at the same physical location, then the monitor will nondeterministically choose an appropriate fileindex/bpindex pair corresponding to some one such type.
4.3.2 Remote Program Environments and the Remote Function Call (RFC) Mechanism

A special feature provided by RXDM is the remote program environments apply/extract feature. This feature is designed to permit the user to modify (using the so-called apply operation) or examine (using the extract operation) in an efficient way the text and data spaces for a given SAP, by means of application of these operations with respect to an environment consisting of a set of bindings. The targeted SAP text and data spaces are actually file abstractions, previously created by means of the OPEN_FILE command, as described above, and are referenced by means of their corresponding fileindex. The environments are maintained by the monitor either in the form of disk files, or the special program environments file abstraction, which is a storage area dynamically allocated in the monitor's memory space via the OPEN_FILE command discussed previously (ref. the special file type OPEN_FILEcmd_PROGENVIR). Each stored environment consists of a simple list or array of bindings, which are tuples of the form <value-type, offset, value>, where value-type is the type of the data object located at the SAP address corresponding to the relative offset from the beginning of the associated text or data space file, and value is the corresponding contents of this location. The value-type must be one of the primitive data types recognized by the monitor communications mechanism, as discussed previously in section 4.2.4 — for example, byte, integer, instruction, etc. Note that the abstract text or data file may have been created as an overlay; therefore, the use of the relative offset in a binding assures the maximum possible machine-invariance of environments (especially regarding abstract data, if not text).

The purpose of the SAP environment apply and extract operations is twofold. First, they are designed to free the user from having to perform a (relatively long) sequence of individual poke (peek) operations, with the accompanying inefficiency due to communications overhead and execution scheduling delays within the monitor. Using the new operations, the user may install or observe variable length lists of mixed-type data items (which he previously created using the appropriate OPEN_FILE command and maintains via the READ_FILE and WRITE_FILE commands) in a single, compact RXDM task.

Secondly, but perhaps more significantly, these operations, used in conjunction with the usual SAP start operations, are designed with the specific application in mind
of implementing a type of abstracted remote function call (RFC) mechanism, similar in concept to the ubiquitous remote procedure call (RPC) mechanism, but slightly different in that the implementation involves the concept of LISP-like application of a function closure to arguments consisting of the bindings provided in an aforementioned environment. The function closure consists of the underlying SAP text plus data spaces. However, the LISP analogy is not strictly correct here, since the environment itself provides identification of the 'variables' whose values will be bound as part of the application, whereas in true LISP function application, the list of locally bindable variables is a fixed part of the closure definition, and consequently can only be altered by a 'partial' application in which some, if not all, of the predefined variables get bound (if not all, then the result is another closure). Here, though, the definition of the validly 'bindable' data locations is removed from the function closure definition and placed into the arguments list itself, therefore permitting dynamic redefinition. Thus the RFC model is one in which the locally bindable variables in the SAP 'closure' are defined as whatever appears in the argument list; all other variables in the closure are considered already bound to values in the SAP data space. In comparison, therefore, the analogy is superficial, and instead what we have in the RFC model is function application based on generalized mutability of locations within the function closure. This actually applies regarding both function data and text (i.e., one can conceptually view text as data, since both are equally mutable).

There are several situations in which the utility of the RFC feature may be exploited. For example, one application is to use these operations to perform run-time initialization of a repetitively-executed SAP (as compared to compile-time initialization), whereby the initializing data may be modified, as need be, from one execution instance to the next using the normal file write operations. In this way, the SAP may be compiled with an uninitialized data (or even text!) area that may be subsequently 'burned in' at the moment the SAP is actually started by the user, using data dynamically derived from another source that was unavailable for inclusion at compile-time for the SAP. In particular, one special case employing this scheme would be intentionally compiling an SAP with blank 'patch areas' included strategically throughout its text space, so that later on, prior to (effective) start of execution, special-purpose text fragments for debugging may be dynamically patched-in (e.g., 'print' statements for interesting variables) using the apply operation (i.e., assuming that the patched-in text fragments were previously stored in the targeted environments files).
A related application is to use the extract operation to capture some relevant part of the data state of a non-running SAP, which could, say, subsequently terminate, and then use the apply operation to reinitialize a new invocation or execution instance of some SAP at a later time or even on a different machine. The latter is facilitated by the fact that the bindings in the captured environment are architecturally-independent (machine-invariant) objects.

A final example of the use of the RFC feature involves just what its name implies: the independent ‘calling’ of text fragments; namely, individual program units (functions/procedures/statements/basic blocks) in a large distributed-parallel program. That is, the user may create (or transport) a ‘function closure’, consisting of a text fragment (such as a basic block) plus an uninitialized data space area, for execution under a monitor by an ‘encapsulation’ process, whereby all the mutable (e.g., uninitialized) referenced variables in the fragment are considered function arguments or free parameters. The function closure is then ‘applied’, on the target machine, to an environment file containing bindings for the uninitialized variables. Finally, after the fragment stops execution (say, due to a special breakpoint located at the end), the extract operation may be used to obtain the results of the function application, perhaps for subsequent ‘application’ elsewhere.

The apply operation is invoked either via the APPLY_PROGTEXT_ENVIR command (in case modifications are to be made to SAP text space), or the APPLY_PROGDATA_ENVIR command (modifications to SAP data space). Likewise, the extract operation is invoked via the EXTRACT_PROGTEXT_ENVIR and EXTRACT_PROGDATA_ENVIR commands. The arguments for these commands include the fileindex identifying the targeted abstract SAP text/data space file, plus fileindices corresponding to the relevant environments (more than one may be specified per operation). Note that since the bindings stored in the environments contain integer offsets with respect to the beginning of the targeted abstract SAP text/data space file (always in units of the stored primitive data type associated with the given binding), references to SAP locations are therefore made in an architecturally-independent fashion. It is also an option with both of these operations that the user can provide an additional global offset, to be uniformly added to the offset for each binding stored in the specified environment files, thereby facilitating relocatability of the bindings in an environment file (useful, for example, with text patch fragments, as touched upon above).
4.3.3 Communications Between RXDM Monitors (*talk* Facility)

A facility is provided with the RXDM monitor that permits interconnection of multiple monitors by means of communications links similar to those that connect the user to a monitor. However, the purpose of these intermonitor communications connections is different, in that they are designed to facilitate two-way communications between different users who are simultaneously running monitors on the same (remote) machine, thereby permitting customized implementation of an equivalent to the ubiquitous UNIX *talk* facility, but in the special context of programming environments supporting computationally intensive programs.

In order to establish contact, both users must be running an instance of the RXDM monitor on the same machine, although both users may actually be based at different remote machines. Therefore, a user who wishes to initiate contact with another user must first determine if the latter is in fact running an instance of RXDM on any of the machines on which he also is running an instance of RXDM. This involves 'exploration' of the machines on the network for the presence of the second user (*i.e.*, an instance of RXDM connected to that user). Of course, it may be the case that the two users have no machine in common. In this situation, the initiating user will have to start up an instance of RXDM on machines not in his working set of machines (*definition*: that set of machines that are each running at least one instance of an RXDM monitor that is associated with the user), as necessary, explicitly for the purpose of probing for (and possibly, subsequently communicating with) the second user.

It must be emphasized that, normally, the initiating user is not really exploring the network for the actual machine on which the second user is based, but rather for any machine that is part of the second user's working set. This is because, by definition, such machines contain an instance of the monitor belonging to the second user through which a communications connection may be established between the two. However, it may be the case that the same user (in the sense of the code used for his identification, or *userid*) is operating separate distributed computational clusters (*definition*: the collection of distributed processes consisting of the root user process (or process group), designated the core, plus all instances of (remote) RXDM monitors and SAP's associated therewith) with each such cluster based (rooted) at a different machine, and possibly even supported on different working sets of (remote)
machines. Because of this, the intermonitor communications facility is actually
designed to permit the initiating user to specify that distributed computational clus-
ter (DCC) to establish a connection with, whereby identification of a particular DCC is
made using the pair $<\text{user\_id}, \text{base\_machine\_name}>$. Effectively, the initiating
user may therefore explicitly name a particular user (ref. \text{user\_id}) based at a particu-
lar machine (ref. \text{base\_machine\_name}) with whom he wishes to establish contact. Al-
ternately, either or both of these may be specified as ‘wild card’, which indicates that
any match is acceptable, as appropriate.

This facility is implemented in the form of special monitor abstract file opera-
tions. Both the initiation of contact by the initiating user and the acceptance of the
connection by the second user are implemented via abstract extensions to the
\texttt{OPEN\_FILE} command, using the special file type \texttt{OPEN\_FILEcmd\_KERNCALL}. After the
connection is established, communication proceeds by means of the usual \texttt{READ\_FILE}
and \texttt{WRITE\_FILE} commands, operating upon fileindices returned by the respective
\texttt{OPEN\_FILE} commands (refer to the discussion of the monitor file system interface in
section 4.2.5). In addition, the special command \texttt{LISTEN\_FOR\_KERNCALLS} is provided,
which permits the enabling/disabling of the ability of a (i.e., the second user's) moni-
tor to respond to attempts by any other (i.e., the initiating user's) monitors to estab-
lish a connection.

In the case of contact initiation, the specification of which DCC to establish a
connection with is supplied by the user as arguments to the \texttt{OPEN\_FILE} command:
the \texttt{user\_id} is placed in the \texttt{file\_name} parameter field, and the \texttt{base\_machine\_name} is
placed in the \texttt{file\_mode} parameter field. The monitor then performs a search of the
locally-resident, so-called Intermonitor Access Directory (IAD), which contains special
files called Intermonitor Access Files (IAF), for an IAF containing information verify-
ing that the local machine is in fact part of the working set for the DCC specified in
the \texttt{OPEN\_FILE} command.

There is a one-to-one correspondence between IAF's and monitors. A unique IAF
is created by a monitor the first time the user issues it the \texttt{LISTEN\_FOR\_KERNCALLS}
command, specifying that the monitor is enabled for responding to attempts at estab-
lishing connections from other monitors running on the same machine. (Note that
any further such disable/enable operations do not affect the IAF itself, rather only
flags internal to the monitor that govern its behavior.) Once created, the monitor's
IAF will exist until the monitor process is terminated.
The IAF for a given monitor contains the information necessary for another monitor running on the same machine to determine if the former is part of a specific DCC; namely, the user identification code (user_id), and identification of the machine (base_machine_name) at which is based the DCC that includes the monitor process corresponding to this IAF. Each monitor's IAF also contains the detailed information required for physical establishment of the communications connection with that monitor process by another monitor process running on the same machine (e.g., under UNIX, this is simply the network address, or port number, for a dedicated listen socket created by the first process at the same time it created the IAF). In summary, therefore, the set of IAF's in the machine's IAD form a database of information about the distributed computational clusters that include that machine in their working sets.

As mentioned, the IAF's for all the monitors running on the machine — irrespective of working set differences — reside in the same common directory, namely the IAD, which is unique in the local file system on that machine. If possible, the IAD is always selected to be a subdirectory of the local temporary files directory (e.g., /tmp in UNIX). This is done in order that the demise of a monitor process, unexpected or not, will always result in the automatic deletion by the file system of the IAF associated with that monitor. Otherwise, if the monitor process were to unexpectedly terminate (for example, due to some as-yet undiscovered bug in the RXDM monitor code that results in, say, a segmentation fault), there would be no opportunity to explicitly delete the associated IAF, which contains information (such as the system process identifier, or pid, for the monitor process!) that is invalidated by the demise of the monitor.

Continuing with the description of the OPEN_FILE mechanism for initiating contact, let us assume that the monitor has searched the IAD and found an IAF that indicates its corresponding monitor is a member of the specified target DCC. The first monitor then uses the information found in the IAF to cooperatively establish a physical connection with the second monitor, and then transmits to it the information identifying the DCC to which it itself belongs. The second monitor then forwards this information to its user in the form of an asynchronous event message, indicating a connection request from the first user/DCC.

At this point, the second user has an opportunity to accept or reject the connection request, based among other things on the identity of the user/DCC initiating the
contact, as supplied in the asynchronous event message packet. If he wishes to accept, he issues an OPENFILE command back to the monitor that sent the event message, again using the special file type OPENFILEcmd_KRNDCALL, only specifying via the command's options a special code indicating that the monitor is to complete (vs. initiate) the establishment of the connection. This (second user's) monitor subsequently will echo back an acknowledgement to the first user's monitor that the connection has been accepted, and then send a reply message back to the second user for the his OPENFILE command, returning therein a fileindex corresponding to the new communications link. The first monitor, which has now received the acknowledgement of acceptance of the connection from the second user's monitor, likewise sends a corresponding reply message back to the first user for the original (initiating) OPENFILE command, returning therein a fileindex corresponding to the new communications link, thereby completing the connection process.

On the other hand, if the second user decides to reject the connection request from the first user, then he can simply perform a close operation (ref. the CLOSEFILE command) on the fileindex corresponding to the pending connection. Alternately, he can simply take no further action in regards to this connection request in order to reject it — the RXDM monitors are designed to automatically timeout the pending connection request, and destroy the (rejected) physical link between them.

Normally, once the intermonitor communications connection is successfully established, the two users may proceed to communicate using the usual monitor file read/write operations (i.e., READFILE and WRITEFILE). In this situation, it is each user's responsibility to periodically and regularly check (i.e., poll) for the presence of data from the other user using the READFILE command; the RXDM monitor will here provide no automatic, asynchronous notification to the user as to the arrival of data on the intermonitor connection (as it originally did for the second user in order to inform him of the connection request from the first user!). The network communications system itself is assumed to provide some sort of FIFO buffering of the pending data on the connection, thereby remedying the strain of rapid polling.

In addition to this polling-type mode, however, the monitor actually does in fact provide the ability for each user to (independently) specify that data arriving from the other user on the intermonitor communications ('talk') link should be automatically and asynchronously detected and forwarded to that user by means of a special asynchronous event message packet (ref. the RECEIVED_DATAMSG event message).
This may be specified as an option in the original OPEN_FILE command performed by each user during the connection phase, or subsequently via the CONTROL_CHANNEL service request directive. The latter actually permits a user random control over the enabling of this automatic event message mechanism, so that he may at one point in the communications select operation in a polling mode, and at another point select the automatic event messages generation mode. Note that these asynchronous ‘talk’ event messages are handled by the monitor in much the same way that it handles asynchronous event messages corresponding to SAP data output (as previously discussed).

Note that either user can, in general, terminate the connection at any time (even during the connection phase, as described) by simply performing a close operation on their respective fileindex corresponding to the connection. Subsequently, if the other user is operating his end of the talk connection in a polling mode, then he will detect the demise of the connection the next time he performs a polling READ_FILE operation. On the other hand, if he is operating his end of the connection in asynchronous event message mode, he will, correspondingly, be automatically notified by means of a special asynchronous event message from his monitor (ref. the FILEIO_DISCONNECTED event message) that the talk connection has been disconnected at the first user’s end.

Figure 4.5 illustrates in general terms the concepts touched upon in the above discussion. The figure illustrates the process by which a communications link is established between two users, 'A' and 'B', assuming user 'A' initiates the contact with user 'B'. Note the dashed lines between the RXDM monitor processes and their associated IAF’s represent the initial creation of these files in the IAD at the time the users first issued LISTEN_FOR_KERNCALLS commands to the monitors.

### 4.4 Application of RXDM within the \( \mathbb{R}^n \) Programming Environment

In this section, I will discuss the specific application of the RXDM remote execution and debugging monitor within the \( \mathbb{R}^n \) programming environment.
4.4.1 The RXDM-FREND Remote Monitor Front-End Interface for Integrating RXDM with R^n's EXMON

As discussed in section 3.3, a 'Remote Monitor Front-End' interface module is required in order to couple the locally-resident R^n EXMON Execution Monitor to
remotely-resident instances of RXDM (the 'Remote Monitor'). Accordingly, a front-end interface module was implemented (under UNIX using C), known as RXDM_FREND, that permits EXMON to dynamically initiate, interact with, and terminate individual remote instances of RXDM. This module consists of a small set of callable routines that effect these functions.

Specifically, the features provided by RXDM_FREND include the following:

- The obvious and necessary mechanism, reciprocal to that implemented in RXDM, for implementing relatively efficient, architecture-independent communications of typed data items in variable-length message packets.

- A mechanism for allowing use, wherever possible, of nonblocking as well as blocking forms of RXDM commands. Basically, when RXDM_FREND is called with the blocking form of a command specified, the call will remain blocked until such a time as the execution of the command either completes successfully, or fails. However, when RXDM_FREND is called with the nonblocking form of a command specified, the call will return as soon as RXDM_FREND has verified that it was successfully received for subsequent execution by RXDM (via handshaking). In such a case, RXDM_FREND will also return a sequence number uniquely associated with the nonblocking command, for future reference. Of course, after the remote command is carried out (successfully or not) by RXDM, then the latter will asynchronously send a reply message back to the local system. RXDM_FREND supplies special routines for dealing with these asynchronous reply messages, as well as other types of asynchronous messages from RXDM (see section 4.2.3). These include a routine for polling the interconnecting communications link for the presence of such a message, and, if present, enqueueing it on a FIFO queue for subsequent retrieval and processing by EXMON. Likewise, a routine is supplied for performing this retrieval function, which pops a stored message off of the top of the queue, and returns the information contained therein\(^4\) in the appropriate format.

\(^4\) which includes the aforementioned sequence number identifying the message
In addition, other utility-type routines are supplied that facilitate the following:

- pre-examining the message on top of the queue without removing it, especially in order to determine its origins;
- cross-referencing a specific, remote instance of RXDM to the channel/file descriptor no. associated with its corresponding interconnecting communications link, and vice-versa;
- cross-referencing logical channel numbers to physical file descriptor numbers for 'direct communications links' to individual SAP's remotely supervised by RXDM (see relevant discussion in section 4.2.3).

- A set of special commands that are endemic to RXDM.FRIEND. These commands are callable in the same way as the usual RXDM commands, and include commands to perform the following:
  - initialize RXDM.FRIEND;
  - shut down RXDM.FRIEND — note this implicitly terminates all existing instances of RXDM, as well;
  - create and initiate a new, remote instance of RXDM — note that the selection of where (i.e., which machine in the distributed computing environment) the new instance of RXDM will be initiated can be optionally specified either explicitly, or left up to RXDM.FRIEND, which will make its decision based on the goal of load balancing;\(^5\)
  - terminate an existing instance of RXDM;
  - implement a special file copy command, that utilizes the RXDM abstract file operations (see section 4.2.5) in order to perform a cross-network file copy — note that, although this command is supplied especially for use in those distributed

\(^5\) that is, 'load balancing' with respect to the relative measure of communications activity being carried out between RXDM.FRIEND and all extant instances of RXDM already on a specific machine
computing environments that did not have some form of NFS available, it is nevertheless unlike NFS, in that with RXDM\_FRIEND copy, all data objects are transported according to their specific type, by means of the RXDM abstract file operations, in an architecture-independent manner (vs. simply in the low-level form of bytes, as with NFS).\(^6\)

### 4.4.2 Implementation Experience and Results

Both RXDM and RXDM\_FRIEND were implemented under UNIX for several architectures, namely the Sun 3 and IBM RT workstations, and the Sequent Symmetry multiprocessor.\(^7\) Note that, since the experimental versions of both of these modules were written in the C language, with UNIX as the targeted operating system, system portability was a fairly straightforward matter. However, since different versions of UNIX were involved, some minor differences in certain UNIX operating system calls required resolution, mainly pertaining to low-level communications operations (in particular, with regards to the handling of file descriptors, as in select()).

Although both RXDM and RXDM\_FRIEND were designed to be self-configuring with respect to some basic architectural properties, such as byte order and integer sign representation, other basic properties, such as word length, code alignment boundary size, and number of registers, had to be manually hard-wired into the respective source code.\(^8\)

---

\(^6\) Furthermore, note that with RXDM\_FRIEND copy, it is possible to perform a copy between any of the various abstract file types implemented under RXDM, as described in pervious sections (e.g., section 4.2.5), whereas NFS is designed for use only with files as defined in the conventional sense.

\(^7\) The Sequent and multiple instances of these workstations form a part of the Rice parallel/distributed computing research network for R\(^3\).

\(^8\) Actually, in order to facilitate maintenance of a single program source for several different architectures, these constants were hard-wired into the source text using the standard C preprocessor (cpp) '#if / #endif' conditional compilation constructs.
Also, a simple test driver was written in order to accomplish initial low-level testing for the basic combination of the RXDM_FRENDF operating in conjunction with a single (remote) instance of RXDM. This driver permitted the testing and evaluation of most of the individual functions remotely provided by RXDM, as well as certain special functions endemic to RXDM_FRENDF (see previous section).

Finally, RXDM_FRENDF was linked to EXMON in the \( \mathbb{R}^n \) system, and this arrangement was exercised and evaluated. However, based on this experience, it was discovered that the resulting system insufficiently dealt with certain aspects of the fundamental abstraction problem connected with a conglomerate computing environment. More specifically, although most aspects relating to architecture-dependence were handled successfully, the system did not fully succeed in this regard since EXMON retained a need to arithmetically manipulate memory address values. Also, the system failed to deal with compiler-dependent variability in the representations of certain higher-order computational objects that are important to the debugging process, such as \textit{stack call frames}. This particular shortcoming consequently motivated the consideration of a completely redesigned, advanced debugging system, as elaborated in the next chapter.
Chapter 5

The Next Generation: An Advanced Abstract Debugging System for Conglomerate Computing Environments

5.1 Introduction

The central theme of this chapter is the discussion of issues pertaining to the design of an advanced abstract debugging system for remote execution and debugging of parallel/distributed programs running in a conglomerate computing environment. After first providing a criticism of the design and performance of the remotely-extended $R^n$ execution and debugging system described in previous chapters, I then proceed to formulate a revised program execution/debugging model that can form the basis for the design of a more advanced and generalized system for remote execution and debugging of parallel/distributed programs.

5.2 Criticism of the Remotely-Extended $R^n$ Model for Program Execution and Debugging of Chapter 3

The RXDM monitor described in previous chapters was specifically designed to provide remote execution and debugging capabilities for that were suitable for an existing system for localized, sequential programs, implemented for the $R^n$ project at Rice (reference the EXMON execution monitor). The associated program execution and debugging (PED) model for this existing basic system was discussed in section 3.2.1.3 and illustrated in Figure 3.3. The PED model for the corresponding system extended for remote domains was described in section 3.2.2.2 and illustrated in Figure 3.5. This latter model formed the motivation for the design and implementation of the RXDM remote execution and debugging monitor (which was designed to integrate with the existing EXMON monitor), and will be the model referenced in the following discussion and criticism.

Although the remotely-extended $R^n$ system was successful in fulfilling many of its design goals, it nevertheless displayed some serious shortcomings, almost entirely
attributable (as it turned out) to the particular _PED model that formed the basis for its design. Namely, the crucially weak factor in this model is that high-level program source code is assumed to be remotely-compiled and directly translated into native object code on the remote machine, although the debugging-related information communicated between the local and remote machines is assumed to be at a lower level of abstraction. This imbalanced situation leads to various types of problems, basically having to do with reconciling configuration-dependent variations (in particular _viz. different architectures and compilers) in the representation of certain debugging-oriented information.

In particular, any remote PED system must be able to map program source-level abstractions present on the local machine, such as variable names and program statements, into the corresponding concrete machine objects on the remote machine. This mapping process is carried out by the 'translate/debug' functions in the PED model. At each level of abstraction, therefore, these functions must have access to a corresponding translation database (labelled DB in the figures) of appropriate cross-reference information. For example, at some level of abstraction, this would include 

< variable name : data memory address > and < statement line number/procedure name : text memory address > pairs, which are normally provided in the symbol table produced by the translation of the program source into machine object code during compilation/linking.

The main problem is, in this particular PED model, the local portion of the debugging system (in particular, the EXMON execution monitor) remains dependent on the remotely-generated cross-reference information found in this symbol table for some of the mappings it carries out. Now, compilers can differ with respect to the

---

1 In addition, as mentioned at the end of section 3.2.2.2, operating system-dependent variations in the mechanisms used for accessing such information and controlling the execution state of the corresponding program is another complication in the design of this, as well as any remote debugging system. However, this particular model does facilitate abstracting away these operating system-dependent parameters at the level of the Remote Monitor (e.g., RXDM), in that relatively little must be changed in a PED system based on this model in order to reconfigure the system to accommodate a new operating system. Thus, this issue will not be discussed further in this section.

2 This shortcoming was touched upon at the end of section 3.2.2.2 as well.
notational form, types, and overall extent of information presented in such a symbol table for a given program, even if purported to be standardized. Also, the memory addresses appearing in a symbol table are usually absolute memory address pointers for the corresponding target machine, which of course can differ in form with different machine architectures. Therefore, because of this retained dependence on remote symbol table information, the design of the overall system is not only very sensitive to variations among the architectures constituting the conglomerate computing environment in which it will operate, but also to variations amongst compilers/programming languages.

In addition, another problem that complicates the mapping process carried out by the 'translate/debug' functions in this PED model is that, again depending on the target machine architecture as well as the compiler, the concrete implementation/representation of certain computational objects\(^3\) for a given program might differ with respect to both their structure and content. This model is sensitive to these variations in the representation of certain computational objects mainly because of the relatively low level of abstraction utilized in communicating debugging information.

Obviously, variations in the concrete representations of certain computational objects can result in the case of differing machine architectures. For example, primitive data types such as floating point numbers have different physical representations — this issue was discussed previously in section 2.4.2. Also, different compilers, even if targeted for the same machine architecture, can produce physically differing representations for a given data structure in a program. For example, this can be the case regarding padding of C-language struct data structures in order to enforce data alignment of individual structure members of a certain data type.

Another especially significant example in this regard has to do with the concrete representation of abstract execution context information on the program run-time stack. Each time a context change occurs during the execution of a program, either as the result of a function or subroutine call, the stack is utilized to dynamically store

\[^3\] as considered apart from the aforementioned static cross-reference information contained in the symbol table
and retrieve information necessary for the context change, normally in a last-in/first-out (LIFO) fashion. The information stored on the stack for each such pending context constitutes a call frame, and includes such items as the contents of all currently active machine registers, the current program counter (i.e., the return address for the call), the current 'program status word' (e.g., special cpu/alu status flags), and any information defining the data environment for computations in the current context. Depending on the programming language as well as the compiler itself, the latter item can include function parameter/argument values (viz. 'bound variables' in the function body), local/temporary variable storage, and possibly some form of frame pointer for dynamically accessing values for 'free variables' in the function body that were bound to values in a previous context. From a debugging standpoint, this information is useful for providing a 'pending' execution history of the recursive calls made to the various functions/subroutines in a program — for example, the ubiquitous stack trace — in order to help determine the source and location of an error, or even to analyze the performance characteristics of the program in terms of the number of calls made to a specific function/subroutine, and the amount of time spent therein. In any case, the code for performing the context changes and associated stack manipulations is always automatically inserted into the compiled machine object code by compilers for high-level languages. Even for the same language, however, the extent and format of the information included in the call frame for a particular function or subroutine in a program can differ from one implementation of a compiler for that language to another. Note that this can even be true for different compilers targeted for the same machine architecture!

Now, this model actually does make provision for the abstracting away of representation disparities for many of these computational objects. This is true, in particular, regarding architecture-dependent conflicts in the physical representation of primitive data types, such as integers and floating point numbers, which can be abstracted in a straightforward manner by the translate/debug functions bridging the 'network interface' translation boundary. In fact, this feature was successfully accomplished in the RXDM monitor (see section 4.2.4).

However, this model does not facilitate an efficient, general solution to the problem of abstracting away variability in the representations of other (especially higher-order) computational objects for a program. This is mainly because of the assumed functionality of the local debugging system (EXMON), which itself is designed for dealing directly with these objects (as indicated by the 'translate/debug' functions
bridging translation boundary "TB2" in the original model of Figure 3.3, and as similarly reflected in the remotely-extended model of Figure 3.5).\(^4\) Consequently, since the mechanisms for dealing with such objects are endemic to the local portion of the PED system (that is, within EXMON), then the local system effectively will be forced to maintain separate knowledge (i.e., in the local translation database) for each unique, remote architecture/compiler/program combination.\(^5\) This complicates the implementation of a PED system based on this model not only from the standpoint of the sheer volume of information that must be maintained at the local level viz. the various machines and compilers resident throughout the distributed computing environment, but also from the related software engineering standpoint of development, installation, and maintenance of such a system. Furthermore, it fundamentally cripples the very concept of an installation-independent remote PED system altogether.

In fact, as alluded to at the end of chapter 4, it was specifically the canonical problem of abstracting away disparate representations of stack call frames in the experimentally implemented system for the \(\mathbb{R}^n\) project, based on the use of the EXMON and RXDM monitors in accordance with the PED model shown in Figure 3.5, that has led to further research in the area of creating an improved, advanced, abstract PED system. As a result of this research, a new remote PED model is being proposed, as described in greater detail in section 5.4 to follow, which reflects a 'downloading' of the responsibility for abstracting away higher-order computational objects — such as call frames — from the local portion of the debugging system to the remote portion. In other words, this new model will involve a more idealized, 'smooth' distribution of the functionality of the remote debugging system, as

\(^4\) For example, EXMON is designed to deal directly with memory address values, which are, of course, architecture-dependent objects. Also, call frames on the run-time program stack pose an especially relevant example in this regard, in accordance with the above discussion, since EXMON is designed to access these frames in order to derive a run-time stack trace. But the information contained in this higher-order computational object is compiler-dependent, and therefore requires access to the remotely-generated symbol table information in order to interpret.

\(^5\) This is largely the reason for the information feedback path shown connecting the remote symbol table (DB on translation boundary "TB3" in Figure 3.5) to the translation database maintained locally within EXMON (DB on "TB2" in that figure).
compared to the models of sections 3.2.2.1 and 3.2.2.2, that should accomplish the fundamental goal of insulating the local portion of the system from architecture- and compiler-dependent variations in the representation of all types of debugging-oriented information, while facilitating relatively efficient communication of such information between machines making up the distributed computing environment.

5.3 High-Level Abstract Debugging of Distributed Programs: Objects and Handles

In consideration of the criticisms presented above, it is clear that the design of a program execution and debugging system for conglomerate computing environments must emphasize the uncoupling of the programmer's view of the abstract data state of each remotely-executing code segment, from the actual, physical data state thereof. Ideally, this will be accomplished through implementation of abstractions that are appropriate for both the programming language being used, as well as the application, or rather class of applications, being handled. These high-level abstractions must subsequently undergo translation into the low-level, physical objects that they correspond to, and vice-versa; this translation process should be performed dynamically by the PED system in a completely automatic way, without burdening the programmer.

Furthermore, the translation process must be as efficient as possible, especially in view of the fact that at least one stage of the process involves communication of relevant information between different computers, that also could have differing architectures. Also, the system should be capable of dealing with optimized code in a reasonable and efficient manner, so there must be some means of translating between optimized and unoptimized code.

One way to accomplish these goals is to design the PED system so that it is based on abstract representations of all elements constituting a program, at all levels except the lowest, corresponding to the architecture-dependent, physical implementation of the program on a machine. However, the abstractions must be suitable for efficient translation, as well as communication between machines. This corresponds to evenly distributing the functionality of the PED system throughout the conglomerate computing environment, with a central, supervisory portion that
interacts with the programmer in terms of the highest level of abstractions, and various remote portions that interact with the remote code segments in terms of the low-level, physical elements constituting those code segments. Connecting the central portion to the remote portions are intermediate levels of abstraction, suitable for dealing with optimizations, as well as intercomputer information transport, in the most efficient way possible. All representations of program elements at this intermediate level must be configuration-independent (i.e., viz. differing architectures, operating systems, and compilers); otherwise, the central portion of the PED system itself will have to have specific knowledge of all the configuration-dependent aspects of these elements, which is undesirable from the standpoint of installation and maintenance of the system in a reconfigurable, conglomerate computing environment.

One approach is to have the intermediate-level representations of program elements map to corresponding high-level abstract objects, which are referenced/identified via shorthand handles. The abstract objects would basically be any element of debugging information that potentially has a architecture-, operating system-, or compiler-dependent representation on a machine. Consequently, this definition corresponds to that given in the previous section for the so-called computational objects making up the abstract data state for a program, including, for example, low-order elements such as machine instructions and the primitive data types (e.g., floating point numbers, memory pointers, etc.), intermediate-order elements such as composite data structures, and high-order elements such as stack call frames.

Note that the handles identifying the objects must be as short and concise as possible in order to facilitate efficient (viz. speed) intercomputer communication, without resulting in inefficient translation to the physical elements being represented. Thus, the choice of the handle for an object is a matter of compromise between these two competing considerations, both affecting overall efficiency of operation of the PED system. Possibly even more significant than this, however, is that the choice of the handle in some situations can place constraints on, or at least parameterize, the overall design and functionality of the PED system, including the compiler that is used to generate the executable program code.

As an example, let us consider what sort of handle might be 'best' in the case of program variables belonging to the primitive data types. At the level of abstraction corresponding to program source code, the variable's name itself could be used as the handle. In this case, the translation of the handle to the corresponding physical object
(i.e., the physical memory location in remote program data space) is very straightforward — all that is required is the usual symbol table information in order to perform the mapping. This translation will take place at the remote computer in this scenario, which means that the symbol table should be available remotely. From this, then, we could infer that the program segment likely would have been remotely compiled on the target machine, which would place a significant constraint on the overall design of the PED system that may or may not be desirable. However, from the standpoint of intercomputer communications, the use of a variable’s name as its remote handle in fact might be undesirable, because in most modern programming languages, names can be any length, and typically can be of such a length as quite possibly to cripple the overall efficiency of intercomputer communications in a large PED system.

In the extreme corresponding to the lowest level of abstraction, the actual memory address of the remote data location could be used as a handle. Certainly, this approach would be desirable in the case of a homogeneous distributed computing environment, in which all machines are architecturally compatible in every way that might affect the concrete representation of program objects; likewise, any compiler-dependent differences in the representations could be forgone by the use of either suitably compatible compilers in the case of remote compilation, or a single compiler in the case all code is compiled locally and then ‘shipped out’ for execution, which is possible since it is given that all the architectures are totally compatible, if not identical. In this situation, intercomputer communications is efficient since addresses are relatively short; similarly, translation of handles to objects is natural, since the handles themselves directly refer to the remote data. However, in the case of an inhomogeneous distributed computing environment, in which the various machines might differ significantly (e.g., differing byte orders and/or word lengths), these advantages are no longer in effect. Thus, the manipulation of remote machine address values can be a problem on the local machine if their architectures differ. For example, in order to locally manipulate incompatible, remote address values as integers, as in performing address arithmetic, then there must be some additional stage of translation to convert the remote values into local integer values; note that this might not even be possible if the remote address value is too large for representing as an integer value on the local machine. In any case, the most fundamental form of manipulation is that the local, central part of the PED system is forced to translate between the remote addresses and the corresponding variable names, which implies that it must have
access to the remotely-generated symbol table (note that compilation obviously could not have been done locally!). This means either that the local system must maintain a local copy of each remotely-generated symbol table (a real problem if dynamic incremental recompilation of the remote code segment is possible), or that it must access the remote symbol tables via intercomputer communications. Although the latter situation might be facilitated via a resident network file system (say), the net efficiency of operation of the PED system nevertheless would be crippled by the continual necessity for remote file accesses. Therefore, from just about any viewpoint, this low-level approach, involving the use of remote address values as handles for the corresponding variables, is flawed both from the standpoint of efficiency as well as complexity, at least for conglomerate computing environments.

Alternatively, at a somewhat higher level of abstraction, the handle could be a unique integer that is associated with each variable name referenced in a remote code segment. For example, all integer variables could be enumerated, beginning with zero; similarly, all floating point variables would then be enumerated, but independently from the integer variables. Alternate, the integer could be prefixed with a single character code identifying the type of the variable being referenced. In any case, this scheme facilitates efficiency in intercomputer communications. However, at least two additional translation stages must be effected — the first of these, located on the local machine, converts between variable names and integer handles; the second, located on the remote machine, converts between handles and corresponding memory addresses. Thus, the gain in communications efficiency brought about by this scheme is to some degree offset by a loss in translation efficiency. Furthermore, since two additional translation stages are involved, this constrains the design of the compiler itself; that is, the compiler ideally should consist of multiple stages, in accordance with the overall translation scheme for the PED system. One such stage of the compiler would be the compiler front end, located on the local machine, which would produce some form of ‘intermediate’ code from the source code, in which all named references to program objects are expressed in terms of handles of this sort. Likewise, another such stage would be the compiler back end, located on the remote machine, which would produce machine language code from the intermediate code, in which all references to program objects are converted from abstract handles into their equivalent concrete representations for that particular machine (e.g., in the case of variables, corresponding memory addresses in the program data space).
A more detailed elaboration of this sort of abstract PED system is given in the following section. More precisely, a new model is presented and developed that would form the basis for the design of an advanced PED system that embodies the principles and features outlined in this section.⁶

5.4 An "Idealized" Model for Remote Program Execution and Debugging

In this section, I discuss a proposed new model for program execution and debugging in distributed, inhomogeneous computing environments. The model is designed to be 'idealized' in the sense that it entails features that, if properly implemented, can help in overcoming the specific problems touched upon in previous sections concerning architecture- and compiler-dependent representation disparities, as well as the general problem of debugging optimized code (both of these are 'mapping'-type problems). However, the model involves not only a revised view of debugging, but also of

⁶ Also, again in reference to the RXDM remote execution and debugging monitor, it is worthwhile to note that this monitor in fact was extended in an attempt to provide a means of architecture-independent manipulation of certain of the lower-order remote computational objects. In particular, the file concept was generalized to permit relativistic manipulation of certain abstract computational objects in terms of the usual file operations. For example, a specific area in a program's data space could be 'opened' for reading and writing; all subsequent references to typed data items in this area then could be expressed (say) in terms of simple integer offsets from the beginning of the 'file'. Therefore, in this scheme, a < file : offset > pair was a sort of abstract handle for remote data locations. However, an absolute address value was still required in order to define the beginning of the abstract 'file' in the program's data space, so really none of the disadvantages described above regarding manipulation of remote memory addresses was completely avoided. Rather, the addition of these generalized file abstractions in the RXDM monitor primarily served to facilitate ease of interaction between the programmer and the remote program objects, as well as to increase the efficiency of intercomputer communications to some degree. Nevertheless, as discussed in the criticism presented in section 5.2, sufficient problems remained with the design of the monitor — or rather, the underlying PED model for which it was designed — to justify additional research and development of a more generalized, abstract PED system. Accordingly, the next section presents a new, idealized PED model that embodies features that overcome the shortcomings encountered with RXDM.
compilation, with the conclusion that any advanced debugging system design must be correspondingly integrated with compiler design, and vice-versa. That is, the model is based on the underlying principle that these two functions are intimately interrelated, not only on a small, single-stage 'translate/debug' scale, but also in the large — so, at any level of granularity, these two functions are basically reciprocals of one another, and this cannot be overlooked nor escaped in the future design of program development environments (such as Rice's R²), that will provide advanced, efficient programming and debugging support facilities.

5.4.1 Description of the General Model

The new model is illustrated in Figure 5.1. Basically, it consists of nine abstraction layers, defined by eight translation boundaries, designated "TBl" through "TB8" in the figure. Each abstraction layer corresponds to a certain level of abstraction, which is mapped to its neighboring layers via 'translate/modify' and 'de-translate/inspect' functions that bridge the respective translation boundaries. The layers range from an applications development environment at the highest level, which contains a specification based on abstract function and data objects, to a run-time environment at the lowest level, which contains the concrete 'machine program' specification based on machine objects such as text and data locations, registers, etc., corresponding to the original high-level specification.

As shown in the figure, the model considers the programmer/user as being capable of interacting with the system at two levels — namely, the applications development environment, and the program source development environment. The former situation corresponds to abstract conceptualization, or visualization, in the most natural way possible, of the abstract data state of a program specifically in terms of those applications-specific constructs or objects that constitute the highest level of specification for the program. For example, if the system is specifically oriented towards development, debugging, and execution of applications that involve the manipulation of queues of geometric figures, then this layer might present the programmer/user with an interface to the system that is based on abstract, graphical manipulation of these objects — namely, pictures representing 'queues', 'boxes', 'triangles', etc. Another, perhaps more interesting example would be the development, debugging, and execution of parallel programs that use Hoare monitors to enforce mutual exclusion in the access of 'critical sections' involving shared data structures.
Figure 5.1: An Idealized Model for Remote Execution and Debugging

In this case, the programmer/user could view and interact with — again, on a graphical basis (say) — the dynamics of operation of the various Hoare monitors, such as which processes get queued into which monitors, and when, all in terms of the objects specifically native to this application — that is, 'monitors', 'processes', 'queues', etc. Note that, in general, this layer must be designed/configured for specific applica-
tions or classes of applications. In particular, research is underway at Rice in the area of abstract visualization of large-scale numerical computations. This research focuses on the abstract graphical visualization constructs or objects that would be most suitable for efficient development, debugging, and execution of this class of applications at the highest possible level, by individuals who are more oriented towards numerical mathematics than towards computer science. Related research in this area includes, for example, the Garden conceptual programming environment being developed at Brown University by Reiss (reference [Rei87]).

The next abstraction layer is the program source development environment, separated from the previous layer by translation boundary “TB1”. The translate/debug functions bridging this boundary are responsible for mapping between the abstract data state of the high-level application objects and the abstract state of the program as expressed in terms of the programming language constructs used in the specification of the program source code. The corresponding translation database for these functions consists of the information required for cross-referencing abstract application objects, such as queues, into the actual, corresponding source-level code and data structures that implement those objects (i.e., abstract data types). The development of these functions, as well as the associated database, is in fact a significant part of the visualization research described above. Also, as mentioned, the model permits the programmer to have the option of interacting with the system directly at the level of the program source development environment, bypassing the application development environment and boundary “TB1”. This alternative corresponds to the conventional approach to source-level program development and debugging.

The next layer in the model is the so-called local interpretive environment layer, based on abstract syntax tree (AST) constructs. In accordance with the discussion of the ‘basic’ and ‘remote’ R^2 models presented in sections 3.2.3 and 3.2.5, respectively, this layer is included in the model to reflect the possibility of selective interpretation (execution) of program code fragments as a technique useful in the debugging process, as explored by Chase and Hood (reference [CH87]). The interpretive execution is carried out by the ‘driver’ function shown in the figure, which thus
effects a simulated evolution of the abstract data state of the program, again for purposes as discussed by Hood. The model provides for this layer to be bypassed dynamically, so that, effectively, the source program development environment can be connected directly with the next abstraction layer, namely the computational environment. Note that the translation boundary "TB2" separating this layer from the previous program source development layer entails a mapping between program source code and equivalent abstract syntax. The associated translation database consists of the information required for this mapping, such as cross-references between source statement numbers and pointers to the roots of equivalent AST data structures.

The computational environment layer corresponds to the next level of abstraction in the model, and contains a specification for the program in terms of an appropriate intermediate language. The purpose for including this layer in the model is the usual one used to justify production of intermediate code by any multi-phase compiler — namely, to loosen and therefore make more flexible the coupling between the source language constructs and the translated target constructs, in order to provide a common and simplified means of representing a program that, first of all, facilitates straightforward application of effective optimization methods, and secondly, facilitates simplified and reliable portability of the translated code into a variety of target execution environments. In relation to this, the particular intermediate language utilized for this layer must be designed to provide a simplified set of operations on certain basic data types that are fundamental to the mechanics of operation of any program. Instances of these basic data types form the abstract computational objects discussed in section 5.1. The important notion in this definition is that the representations of these objects must be standardized and uniform, abstracted away from displaying any implementation or installation dependencies, as connected with a particular network, architecture, compiler, or operating system. Trivially, of course, data variables and constants with the usual primitive data types, such as integers, floating point numbers, etc, are included in the definition, so the physical characteristics of their implementation on a particular architecture, such as byte order, sign represen-

7 For an explanation of one of these purposes, see section 3.2.1.3 and the associated relevant footnote.
tation, exact number of bytes of precision, etc., must be abstracted away at this level. A similar situation obviously holds for pointer variables and function/subroutine references, as well. Likewise, another such abstract computational object that is particularly relevant to debugging is the abstract and uniform representation of procedure call frames on the (dynamic) program run-time stack, at least for the purpose of tracing the pending execution history of the recursive calls made to the various functions/subroutines in a program. Section 5.4.3 discusses the design of a particular candidate intermediate language in greater depth. Furthermore, note the translate/debug functions bridging the translation boundary “TB3” that separates this layer from the source program development environment layer. The translation database associated with these translate/debug functions — which perform a mapping between source code and intermediate code — contains all information appropriate to the mapping, such as cross-references between source statement numbers and equivalent blocks of intermediate code.

The next abstraction layer is the optimized computational environment, which contains an optimized version of the intermediate-code specification from the previous layer. The optimized version is still expressed in terms of the same intermediate language constructs. The translation boundary (reference “TB4”) separating this layer from the previous layer is bridged in the forward direction by a ‘translate/modify’ function that acts to effect the optimizations. A corresponding ‘de-translate/inspect’ function bridges the boundary in the reverse direction, mapping the abstract data state of the optimized constructs into that for the equivalent unoptimized constructs. Note, however, that this inverse mapping may not be possible in every case, since some optimizations result in such profound changes to the original code that the inverse mapping process is completely ambiguous. A partial solution to this problem might be to perform the optimizations in small ‘increments’, spread out over many sub-stages or layers, rather than all at once as in the single-stage scenario shown here. That is, this would entail replacing this particular abstraction layer with several sub-layers, plus associated translation boundaries — although the same net, overall optimizations would be effected, the scope or magnitude of the individual optimization increments performed as mappings across each such new translation boundary would be thus much more limited. Consequently, this modified model would retain cross-referencing information at a more detailed level, corresponding to intermediate optimization steps that otherwise would have been factored or condensed out in the overall mapping, thereby in many cases facilitating the previously unattainable,
unambiguous inverse mappings between optimized and unoptimized code.

The next abstraction layer, the machine front-end transport environment, is based on the use of constructs from a somewhat lower-level language for standardized intercomputer communications, or intercomputer transport language. This language is similar in definition to the previously-described intermediate language, except that all constructs are essentially codified into a special format syntax in order to facilitate efficiency of communications of (abstract data state) information between machines, across a distributed network. For example, symbolic name references — such as function, subroutine, and data structure names — for computational objects in the intermediate language code, which can have an unspecified length, are relatively inefficient candidates for communicating across a network. Similarly, the concrete syntax of the operation constructs in the intermediate language might be relatively inefficient for communicating over a network. Therefore, this layer implements an abstract representation of these objects via concise aliases or codes. In particular, in the case of computational objects, these codes correspond to relativistic, typed handles for distinguishing the individual objects. For example, the integer variable named ‘z’ might be mapped into the handle ‘117’, to indicate the ‘17th’ member of the list of all integer variables for the program. Also, procedure names would be viewed as positional labels; that is, offsets, expressed in terms of the number of operations relative to the start of the code. Thus, procedure names, as well as normal labels, could be assigned relativistic handles in a manner similar to variables. As another example, the most recent (viz. real-time execution) call frame on the runtime stack for any program might be assigned the standard handle ‘$0’; the previous frame would then be designated (say) ‘$–1’, and so on, whereas the oldest (first) frame on the stack could be assigned the special handle ‘$1’, with the subsequent frame designated ‘$2’, etc. Also, individual data items constituting a frame could be referenced in the usual hierarchical ‘member’ fashion — e.g., the handle identifying the procedure that was invoked in the next-most-recent frame would have the handle ‘$–1.proc’ (say), and the return address (program counter, or ‘pc’) stored in that frame for returning from that procedure would have the handle ‘$–1.pc’ (say). Consequently, the translation boundary (reference “TB5” in the figure) between this abstraction layer and the previous layer basically involves only a straightforward mapping between symbolic names in the (optimized) intermediate language, and these relativistic, typed handles.
The next layer, the *machine back-end transport environment*, is defined essentially in the same way as for the front-end layer. That is, this layer contains the same form of representation for the program as did the front-end layer, expressed at the same level of abstraction and with the same intercomputer transport language constructs. The only difference is that this layer resides on a different machine, so one might view the previous layer as containing a 'local' image of the abstract representation contained in this 'remote' abstraction layer, or vice-versa. More interesting than this, however, is the 'NETWORK INTERFACE' translation boundary (reference "TB6" in the figure) that separates the two layers. The translate/debug functions bridging this boundary actually involve several implicit levels of functionality, all related to the physical communication of information across a distributed network of machines with possibly inhomogeneous architectures. Thus, their purpose remains the same as that described for the model in Figure 3.5 — that is, to implement physical transportation of the program information represented using constructs from the intercomputer transport language. This is accomplished via appropriate construction, transmission, reception, and interpretation of concrete message packets. Note that this transport mechanism also involves any necessary reconciliation between disparate, architecture-dependent representations of primitive data type values, such as differing byte orders for integers, as previously discussed. In other words, it is during the transport of data across this particular translation boundary that any architecturally dependent representation conflicts for low-order program computational objects, such as the primitive data type values, are resolved. On the other hand, note that representation conflicts for 'values' of high-order abstract computational objects, such as stack call frames, are resolved at the next translation boundary, as described below.

The next abstraction layer is the *machine-optimized environment*, which contains a translation of the program representation based on the intercomputer transport language code of the previous layer, into a representation based on the native language (assembly code) for the given machine, that also has been optimized to take advantage of the architectural characteristics of the machine, such as the total number of registers available. Note that the translation boundary separating this layer from the previous layer (reference "TB7" in the figure) serves the important purpose of distinguishing the previous environment, containing abstract computational objects and intermediate-level code operations, with all references in terms of handles (which actually are high-level abstract objects), from an architecture-dependent,
low-level environment containing machine language specifications, with all references in terms of machine addresses/labels. Thus, the translate/debug functions that bridge this boundary perform a mapping between abstract computational objects for a program, such as stack call frames, and low-level machine language specifications that are optimized with respect to the specific machine architecture. These machine language specifications in turn describe concrete machine objects, such as text and data locations, registers, etc., which are contained in the next and final abstraction layer, the run-time environment (see below).

Actually, the 'translate' function bridging the boundary in the forward direction corresponds to the conventional compiler back-end concept — it translates what amounts to intermediate-level, architecture-independent code into equivalent architecture-dependent, native machine language code. Accordingly, this compilation process generates the associated translation database, which corresponds to the conventional back-end symbol table, except the 'symbols' involved are handles, versus the usual symbolic names, for describing the program's computational objects. Therefore, the database contains cross-reference information appropriate for the inverse mapping effected by the 'de-translate' function. For example, it contains that information necessary for translating the current value of the program counter (an absolute memory address) into the handle for the specific, corresponding procedure that is currently being executed. In a similar way, the return address (also a program counter value), as stored in a particular call frame identified by its associated handle, can be used to look up the handle for the procedure corresponding to that frame, that was once being executed as the result of the (pending) call.

The last abstraction layer in the model is the physical run-time environment, which contains the concrete representation of the actual, low-level abstract data state of the program in terms of the contents of physical machine objects, such as specific text and data locations in memory, and machine registers. The translation boundary separating this layer from the previous layer (reference "TB8") is bridged by translate/debug functions that map between native machine language constructs and the equivalent, physically implemented machine objects. The associated translation database contains the corresponding cross-reference information, such as <address label: absolute memory address>.

In summary, the approach taken in this model, whereby all architecture- and compiler-dependent mappings occur across translation boundaries located only on the
remote machine, provides a particularly effective solution for the abstraction problem.*viz.* representation disparities for certain computational objects, as discussed in section 5.1. That is, such an abstraction problem results from the situation in which differing architectures and compilers are used in a conglomerate computing environment. The solution is especially efficient from the standpoint that the relevant translation boundaries are located on the remote machine, so the centralized, supervisory part of the program execution and debugging system, located on the local machine (*viz.* the programmer/user), deals only with a fixed level of program abstractions, and is not burdened with maintaining installation-dependent information for all the unique, remote architecture/compiler combinations extant in the distributed system.

5.4.2 A Constrained Version of the General Model that is Suitable for Practical Implementation

For a practical implementation of a program execution and debugging (PED) system corresponding to the above model, it may be desirable to constrain the aforementioned 'intermediate language' and 'intercomputer transport language' to be the same language. Of course, if this is done, then the optimized computational environment and the machine #1 front-end transport environment effectively become the same, since the translate/debug functions bridging the translation boundary between the two environments (reference "TBS" in Figure 5.1) only serve to provide conversion between representations of program objects as expressed in the two languages.

The revised version of this model, in which the two languages have been constrained to be the same, is shown in Figure 5.2. In addition, note that this model reflects the addition of a new 'driver' function at the level of the back-end transport environment. The purpose of this driver is to effect remote interpretive execution of the abstract, unoptimized intermediate/intercomputer transport language code, as an optional alternative to the normal execution of the low-level machine code in the run-time environment. Thus, as for the local interpretive environment described in the previous section, this model provides for this driver to be bypassed dynamically, so that, effectively, the back-end transport environment can be connected directly with the next abstraction layer, the machine-optimized environment, for 'normal' execution.
Figure 5.2: Constrained Idealized Model for Remote Execution and Debugging in which Intermediate and Intercomputer Transport Languages have been Merged

Note that, in general, optional interpretive drivers actually could be situated at any (or all) of the environments in this model, either locally (excluding, of course, the local interpretive environment, which already is interpretively-driven) and/or remotely. However, for various reasons, only one additional such optional interpretive driver was utilized in this revised model, and was specially situated at the back-end transport environment layer. One important reason for this particular choice was to
permit optional bypassing of the need to perform mappings between unoptimized and optimized versions of code\(^8\), which can be quite a difficult task in some instances. Also, locating the driver remotely, as indicated, was deemed more appropriate than locating it locally, such as in the computational environment or the optimized front-end transport environment, since that would defeat the obvious benefits (viz. speed and space requirements) derived from distributed execution. Another important consideration in this choice was the practical need to facilitate rapid future development and implementation of a PED system based on this model. That is, the interpretive driver, situated as indicated, can function as an architecture- and machine-independent, temporary substitute for the remaining, installation-dependent, low-level components in the model, until such time as they would be developed and implemented.

5.4.3 A Canonical Intermediate/Intercomputer Transport Language for the Constrained Version of the Model

A canonical language for the combined intermediate/intercomputer transport language assumed in the revised, constrained model of the previous section, must obviously combine features from both original languages, including constructs that facil-

---

\(^8\) Namely, in this model, one such mapping is performed locally, across translation boundary "TB4" (see the figure), connecting the computational environment abstraction layer to the optimized front-end transport environment layer. Another such mapping is performed remotely, across translation boundary "TB5", connecting the back-end transport environment and the machine-optimized environment layers. (Actually, note that the layers associated with optimized code might consist of multiple sublayers, in order to facilitate the mapping process for optimizations, as mentioned in the discussion in the previous section of the (local) optimized computational environment for the original model.)

The location of the interpretive driver at the remote site was favored in accordance with maintaining the obvious benefits of distributed execution. Obviously, however, any problems associated with the mapping of optimizations across "TB4" remain in force. One solution to this dilemma would be to constrain the design of any system based on this model in such a way that the particular optimizations performed locally across this boundary would be especially straightforward viz. mapping. Alternatively, the model could be changed to include another interpretive driver situated in the computational environment, specifically to intercept and handle just those situations in which the mapping problem is 'too difficult' — the remainder would be passed on to the lower abstraction layers, as usual.
iterate various types of optimizations, as well as efficiency of communication of abstract data state information between differing machine architectures, in an architecture-independent form. The usual approach used to achieve the former is to define the intermediate language constructs as simplified operator/operand *triples* or *quads*. Also, the latter goal would be achieved by the appropriate use of encodings, or handles, for abstractly referencing all the computational objects in a program in an efficient yet installation-independent way (as described in the previous discussion).

In addition, the ideal canonical intermediate language should be augmented with extended operations that are specifically intended for facilitating remote program execution and debugging.9 More specifically, these operations include the following:

(1) The usual collection of commands for controlling the execution state of a program — namely, the following:

(a) a *start* command for initiating execution of a program;

(b) a *suspend/pause* command for temporarily halting execution of a program;\(^\text{10}\)

(c) a *single-step* command;

(d) a *resume/continue* command for restarting execution of a previously *suspended* program; and,

(e) a *terminate* command.

(f) Furthermore, ideally, the *single-step* and *resume/continue* commands in particular should be designed to permit specification of the *direction of execution* that is desired — i.e., either forward (normal) or reverse.

(2) A set of commands could be provided specifically for remote manipulation of breakpoints as a *type of abstract computational objects*; for ex-

---

9 For instance, consider the *DELAY* construct suggested by Kaiser [Kai85] for implementing breakpoints, etc.

10 Note that execution of the *suspend/pause* command should force the automatic reporting of the corresponding event to the higher abstraction layers in the PED model, for notification purposes. Also see the discussion of 'update strategies' for debugging-oriented information, in section 2.3.1.
ample:

(a) a set_brkpt command for installing a breakpoint into program code; and,

(b) the complementary delete_brkpt command.

Note that these operations could work with various types of breakpoints, such as temporary or conditional breakpoints. Similarly, a set of commands could be supplied for artificially adjusting the synchronization of parallel/distributed code segments, such as:

(c) a set_synch command for installing an enforced synchronization point into a parallel code segment; and,

(d) the complementary delete_synch command.

Note that these synchronization points could be implemented using a scheme based on distributed breakpoints, as described in Miller and Choi [MC88b].

(3) The usual collection of commands for directly examining and modifying the abstract data state of a program — namely:

(a) a examine/inspect command for examining the contents of either the text or data spaces of a program; and,

(b) the complementary modify command.

However, it is important that these particular commands be geared for a level of abstraction appropriate to the use of installation-independent references to the abstract computational objects constituting a program, such as the handles previously described in section 5.3.

(4) A set of commands specifically intended for dynamic internal restructuring of the remote program code, either as a result of incremental recompilation, or dynamic insertion/removal of code fragments viz. the existing remote program code, primarily for debugging purposes. These commands might include the following:

(a) a create command for physically creating (via incremental compilation techniques) a code fragment to be inserted into existing program code;
(b) an *insert* command for performing the installation of the created fragment (i.e., a code *patch*);\(^{11}\)

(c) a *remove* command for deactivating\(^{12}\) a previously inserted fragment; and,

(d) a *destroy* command for eliminating the definition of a fragment.

Note that the *insert* command will likely result in some physical restructuring of the low-level machine code for the program; likewise, the *remove* command will restore the original code.\(^{13}\)

(5) In addition to the above, there could be other commands for implementing generalized surgery on a program's text and data spaces. Among these would be:

(a) a *set.bypass* command, which would effect a jump around a specified section of existing code in text space; and,

(b) the complementary *delete.bypass* command, for removing a previously set bypass.

---

\(^{11}\) More specifically, this installation process normally would involve the following sequence:
- given the existing instruction at the location in the original text at which the fragment will be inserted, append a copy of this instruction at the end of the fragment body (which is situated in some 'patch area', with suitable blank space assumed to be available);
- replace this instruction in the original text with a *jump* instruction that has as its operand the location/address of the beginning of the fragment body ('jump-to-patch');
- append a *jump* instruction to the (revised) end of the fragment body, that has as its operand the address/location of the instruction in the original text that is located immediately after the previously-copied instruction ('return-to-prog');

\(^{12}\) That is, 'deactivation' of a previously inserted code fragment corresponds to reinserting the *original* code displaced by the fragment as a result of the installation process described above, so that that particular instance of the fragment no longer is reachable for execution. However, the contents of the fragment are not disposed of as a result of this command, since once it is *created*, a copy of each such unique code fragment is stored for later use, as required, until such time as it is *destroyed*.

\(^{13}\) Refer to section 5.5 for additional discussion of the issues pertaining to dynamic program restructuring.
Other useful 'surgical' commands would be:

(c) a copy command, which would create a fragment whose contents are a copy of a specified section of either the program text or data space;

(d) an excise command, which also would create a fragment whose contents are a copy of a specified section of either the program text or data space, but would then automatically 'blank out' that section — in the case of text, this would be accomplished via replacement of the existing code with 'noop' instructions; in the case of data, replacement would be made with zeroes;\(^{14}\) and,

(e) a substitute command, which would be the complement of the excise command, and would replace the contents of a specified section of either text or data space with the contents of a fragment.\(^{15}\)

---

\(^{14}\) Note that the semantics of this particular command must be carefully considered. In particular, it is probably desirable that the command should not be allowed to corrupt the existing program in unexpected ways, that subsequently are difficult to deal with. For example, in the case of application to program text space, the command could be designed to fail in the situation part of a control flow selector instruction (e.g., the 'if..then' portion of if cond then \(S_1\) else \(S_2\)) occurs within the specified section to be blanked. Thus, this command should only be effective in the case its potential result is syntactically consistent.

\(^{15}\) The semantics of this command must also be carefully considered. For instance, if the extent of the specified section is too small to hold the entire contents of the fragment, then the command could be designed to fail. Alternatively, the command could be designed such that, in this case, a new fragment would be created, whose contents are a copy of the remainder of the original fragment. This new fragment then would be inserted at the end of the specified section, in order to effectively complete the substitution. Of course, this entire process could be performed automatically. However, the substitute command probably ought to fail if the specified section already contains a previously inserted fragment, in order to avoid the complexities involved with substituting into such a fragment. The same holds for the extract command, as well. In fact, the substitute command should probably fail under all the same circumstances under which the extract command fails, and vice-versa (see the previous footnote viz. the extract command).
Finally, it would be useful to have commands that can effect random dynamic initialization of a program's text and/or data spaces. These commands could include:

(a) an apply command for substituting the values contained in a list of typed fragment bindings into the existing program text or data space, as appropriate; and,

(b) the complementary extract command that would copy values from program text and/or data space into a list of typed fragment bindings.

Now, certain constraints apply in the design of an intermediate/intercomputer transport language that could suitably support the operations outlined above. For example, in order to efficiently support the code fragment manipulation commands in the context of dynamic program restructuring, the language ought to display a so-called 'locality of change' quality — namely, given the representation of a program consisting of constructs from this language, then small, localized changes to this representation in terms of addition or deletion of a few instances of the constructs should not adversely perturb the semantics of the remainder of the representation. A negative example would be the inclusion of some form of relative jump instruction in the language; if code is inserted or deleted in the region jumped over, then the

---

16 Note that implementation of a similar concept was described in section 4.3.2.

17 A typed fragment binding is an object defined via the following tuple: \(< \text{space}, \text{section}, \text{type}, \text{fragment} >\), where \(\text{space}\) is either \(\text{TEXT}\) or \(\text{DATA}\), \(\text{section}\) defines an area in this space, and \(\text{type}\) defines the 'type' of the fragment referenced in \(\text{fragment}\) (e.g., \text{DOUBLE_PRECISION} in the case of a data fragment whose contents are a single double precision floating point value). The 'value' contained in this binding is the indicated fragment. The apply command effectively performs an implicit substitute command for each binding in its argument list, with appropriate consideration given to the 'type' of the indicated fragment — the arguments required for each such substitute command are taken from the binding. Note that I am also assuming the provision of the obvious auxiliary commands for manipulating these bindings, as well as lists thereof ('environments').

18 That is, the typed bindings would already be defined, at least with respect to the location information contained in the binding — namely, \(\text{space}, \text{section}, \text{type}\) —. An implicit copy command is performed for each such binding, using these values as arguments, which creates a new fragment. The binding is then updated to reflect this new fragment by placing a reference to it in \(\text{fragment}\).
operands of the *jump* instruction would have to be changed in order to compensate for the code modification, so such a language likely would not be suitable.

In particular, one suitable candidate for the combined intermediate/intercomputer transport language could be an appropriately extended version of the ILOC [BCT87] intermediate language being developed for the R^n project at Rice. Note that ILOC largely satisfies the above 'locality of change' quality. Also, an optimizing FORTRAN compiler already has been developed for this project that produces optimized ILOC code from FORTRAN source code. Also, a compiler for the Russell language has been developed in separate research at Rice, which also produces ILOC intermediate code. Thus, these existing compiler front ends would facilitate future research for an advanced program execution and debugging system.

### 5.5 Dynamic Program Restructuring via the Advanced Abstract Debugging System

The empirical debugging cycle discussed in section 2.2.5 was based on repetitively carrying out the following tasks: examination of the abstract data state of a program; comparison of the observed data state with the expected data state; and finally, modification of the abstract data state, in accordance with any differences measured in the previous task, in order to implement a 'correction' to the program. In accordance with the definition of abstract data state, these modifications can affect the text/instructions as well as the data structures that constitute the program. Therefore, the effect of the empirical debugging process is an evolutionary, dynamic restructuring of the existing program (or rather, its abstract data state) in order to bring it into conformance with the envisioned, 'correct' version of the program.

Consequently, it is clearly desirable to be able to carry out this dynamic restructuring of the program at the highest possible level of abstraction that is appropriate to the kinds of errors appearing in the program. In particular, the handling of textual modifications ought to be facilitated at the level of the original source-level programming language constructs. This means that high-level textual modifications eventually must be propagated or translated into their equivalent, low-level, concrete machine code representations, in order to carry on with the debugging cycle. In conventional programming, this corresponds to modifying the original source text specification,
recompiling the entirity of this modified specification, and then relinking the generated object code into an executable, machine code form, which finally is then re-executed. However, it clearly would be preferable to have all this performed automatically, specifically only with respect to the modifications themselves, without involving the remaining entirity of the original program specification. Thus, the existing, executing program would remain in place, being disturbed only to the extent required for the physical implementation or installation of the translated textual modifications. This approach corresponds to incremental recompilation of the textual modifications, in which only the modifications themselves undergo translation from source code level to machine code level, rather than the entire program text, and are subsequently installed into the existing machine code program in an automatic and otherwise inobtrusive fashion. The following subsections discuss incremental recompilation in more detail, at least insofar as it applies to the design of an advanced program execution and debugging system for conglomerate computing environments, such as that based on the model described in section 5.4 above.

### 5.5.1 Incremental Recompilation

In general, almost all debugging systems provide the ability, at some level of abstraction, to modify the text and data spaces of an existing, already compiled program, without recompiling the entire program. Usually, however, such modifications are restricted to simple replacements of the contents of corresponding, low-level memory locations. For textual replacements, the level of abstraction that is involved is thus no greater than that inherent in low-level machine language instructions. However, for data replacements, the level of abstraction is usually somewhat higher, corresponding to that of the primitive data types, such as integers, floating point numbers, etc.

Thus, these replacements are equivalent to a form of incremental recompilation, at least at the level of machine language constructs. In constrast, an advanced debugging system should facilitate modifications, particularly textual modifications, at the highest possible level of abstraction — that is, minimally at the level of the source programming language itself.

Basically, there are two approaches for accomplishing modifications at this higher level of abstraction. The first, and most frequently used, is based on maintain-
ing a separate, **interpretable** representation of the program that is semantically equivalent to the original source-level code, in order to provide a means for facilitating post-compilation modifications to this code. That is, modified regions of source code, such as dynamically inserted statements, can easily be executed interpretively, while the unmodified regions, which were previously compiled into (remote) machine code, can be executed normally. Actually, this scheme effectively bypasses altogether the alternative requirement that some method of incremental recompilation be used to handle the textual modifications.\(^{19}\)

However, there are at least two major drawbacks with this scheme. One drawback is the obvious inefficiency involved with carrying out interpretive execution — interpretation is generally much slower than direct execution of compiled machine code. In fact, this is an especially relevant consideration in terms of synchronization effects, as discussed in section 2.4.1! For example, modified code that seems to work fine using this interpretive scheme may not work at all when recompiled and reexecuted, because the differences in speed of execution between the two situations can affect the timing relationships of the various cooperating parallel processes.

Another drawback is that, since the interpreted code is executed independently with respect to the compiled, machine language code, then the abstract data states for the two execution instances need to be *updated* in order to reflect the changes made to data structures whose contents can be affected by either execution instance. In other words, the abstract data states for both instances must be *consistent*, at least at the point immediately before the execution of an instance begins. This update process consequently entails the additional inefficiency inherent in the manipulation of information, which again could be reflected in anomalous synchronization effects.

Note that particularly appropriate constructs for expressing the interpretable representation are abstract syntax trees (AST's), as indicated in the description of the 'idealized' model for remote program execution and debugging presented in section 5.4.1 (also reference the AST-based *local interpretive environment* abstraction layer, as shown in Figures 5.1 and 5.2). The idea is that AST's are relatively easily to modi-
fy, and especially since they are being interpreted, the effects of the modifications can be immediately realized, without involving recompilation of the entire representation into concrete machine code. This argument, in fact, forms the primary basis for including

The second approach does not rely on the crutch of interpretation, and is based on restructuring all lower-level representations in accordance with modifications made at the source level. (One such system based on this approach is the DICE system described by Fritzen [Fri84].) For a multilayer abstraction model such as that of Figure 5.2, the fundamental problem is one of correctly mapping the modifications between the successive levels of abstraction, in order to propagate them from the highest level (i.e., either the application development environment, or the source program development environment) to the lowest level (i.e., the machine run-time environment, consisting of concrete machine objects), in a sort of pipelined fashion. Furthermore, there are two special aspects of this mapping problem — namely, how to deal with optimizations, and how to create 'room' for new code and data objects, if needed, especially at the intermediate and lower levels.

In regards to this latter issue, one option might be to intentionally pad the original, intermediate- and low-level representations with blank 'patch areas', suitable for holding future modifications. For example, the compilation process, or rather the 'translate' function connecting each stage of abstraction, might automatically generate a fixed size, blank 'in-line' patch area before and after each emitted instruction, which either could be filled with 'no-ops', or bypassed via an embedded jump. The problem with this is, of course, that these blank areas may be insufficiently sized with respect to the actual size of future modifications. Alternatively, if the modifications can be accomplished via code fragments, then these can be 'grafted' into existing code via the insert command described in section 5.4.3 under item (4), without the need for such 'in-line' patch areas, since the insertion process (see the relevant footnote in that section) only involves the substitution of a jump instruction for the existing instruction located at the point of insertion. Nevertheless, patch space still must be available somewhere for holding the body of the code fragment itself.

---

20 Unless, of course, some of the modifications result in the deletion of some of the original instructions, thereby freeing up additional space in sufficient quantity.
With regards to a multilayer abstraction model such as that of Figure 5.2, note that incremental recompilation of modifications, in particular a code fragment corresponding to new code to be patched into existing code, can be effected in a pipelined fashion, as part of the operation of the 'translate/modify' functions bridging the translation boundaries separating the various abstraction layers. Consequently, special command constructs may be supplied, particularly for the intermediate and/or intercomputer transport languages referred to in this model, that facilitate the manipulation of these code fragments (e.g., the aformentioned insert command). A summary of such commands was previously given in section 5.4.3, in the discussion of a possible canonical form for these languages. Also, the next subsection elaborates upon the implementation and application of these special command constructs in the context of utilizing inserted code fragments as a specialized debugging technique.

5.5.2 Dynamic Insertion of Code Fragments into Existing Program Text as a Specialized Debugging Technique

There are basically two classes or types of debugging systems, categorized according to whether the debugging system is incorporated into the program code at compile time, becoming a direct, integral part of the code itself ('code-resident'), or operates as a separate entity ('non code-resident'). Debugging systems in the first class are usually simple in terms of their implementation and provide a high degree of resolution in the observation of the evolving abstract data state of a program. However, they are highly invasive, and consequently have the potential for affecting the observed behavior of the program, especially in regards to timing or synchronization effects. On the other hand, debugging systems in the second class are largely non-invasive, but nevertheless can affect the observed behavior of the program in unexpected ways, although most likely to a much lesser degree than with the code-resident class of debugging systems.

The most basic example that can be drawn from the class of code-resident debugging systems is the ubiquitous method of explicitly including the print statement within one's program at various, strategic locations. There are usually two reasons for doing this: first, to log or trace control flow, such as revealing whether execution of the corresponding point within the program has been reached, or is reachable, as well as the relative order of execution of several such points; and, secondly, to trace data flow, such as displaying the evolving contents of certain critical data structures,
whose reported values then may be compared to expected values in order to detect the occurrence of an error. In this way, the print statement is used as an embedded, fixed probe into the abstract data state of a program, as part of the empirical debugging cycle.

Likewise, a non code-resident debugging system can be used to dynamically insert specialized code fragments directly into the executable instructions/text space of an already compiled program, therefore in effect permitting the dynamic installation of code-resident debugging features, such as high-resolution tracing. These code fragments, of course, themselves must be in ‘executable’ form, which implies that they must have been incrementally compiled. The process of inserting a code fragment could be effected via the usual inspect and modify commands supplied with any debugging system, which are used to examine and mutate the existing text and data spaces of a program, assuming that sufficient space exists in the existing program code for holding the new code fragment.

However, as mentioned in section 5.4.3 for an advanced debugging system, it would be better to provide special, somewhat higher-level commands for manipulating code fragments, such as create, insert, remove, and destroy, as constructs in the architecture-independent, intermediate/intercomputer transport language (IIITL) associated with such an advanced system. One good reason for this is that these commands could be easily incorporated into the code fragments themselves, thereby permitting the design of powerful, self-modifying code fragments that are particularly useful for debugging.

As an example, these commands can facilitate implementation of temporary breakpoints, which are a type of breakpoint designed to act only once. This type of breakpoint is sometimes used to implement a 'run to here' debugging function.

First, I will refine the semantics of the remove command such that, if it refers to the same code fragment in which it itself resides, then that particular fragment is not deactivated until control flow actually exits the fragment. Otherwise, if it refers to another fragment, or if it does not reside in a fragment at all, then the specified target fragment is deactivated immediately. Note that this definition will also affect the semantics of control flow transfer constructs in the language, such as jump.

Thus, a code fragment that will implement a temporary breakpoint would be centered around the following simple IIITL instruction sequence:
F1: remove this-fragment; /* set up automatic deactivation of this fragment */
    suspend; /* temporarily suspend/pause execution */
< original instruction displaced by insertion of this fragment >;
jump F2;

where the modified program text looks like

... 

< previous instruction in the original sequence >;
jump F1; /* location of displaced instruction */
F2: < next instruction in the original sequence >;

... 

After execution returns from the suspend, the remainder of the code fragment will execute in accordance with the above definition of remove, including the original, displaced instruction. Likewise, the jump instruction at the end of the fragment will first cause the fragment to be deactivated before execution actually resumes at statement label 'F2' in the program text area (line no. 7). That is, the original, displaced instruction (ref. line no. 3) is restored to its original position at line no. 6, replacing the jump instruction placed there when the fragment was first inserted. Normal execution then resumes at line no. 7. Consequently, if this same program text area were reentered, then the fragment would not be active. Thus, the temporary breakpoint has evaporated after its first execution instance, as desired, thanks to the self-modifying qualities of the special constructs used in the code fragment that implemented it.
Chapter 6
Summary and Conclusions

Overview

Historically, the mechanism for program execution and debugging in the Rice $R^n$ programming environment (i.e., the EXMON execution monitor) was originally designed and implemented for local use — that is, for use with programs running on the same processor as this PED mechanism. Of course, the standing goal of the $R^n$ system is to be generally applicable in a distributed computing environment, supporting execution and debugging of large-scale scientific parallel/distributed programs. Therefore, in accordance with this goal, the development and implementation of a means of extending the existing $R^n$ PED mechanism (i.e., EXMON) for remote operation in a distributed computing environment was proposed by Chase and Hood ([CH87]). Furthermore, this proposed scheme was intended to remedy several of the drawbacks associated with various existing source-level, remote PED systems (canonically exemplified by the well-known dbxtool system) in which an individual instance of a standalone PED system (e.g., dbx) is associated with each remotely-resident code segment constituting the parallel/distributed program. Such drawbacks include:

- the difficulty involved in adapting such systems to various installation-dependent combinations of machine architectures, compilers/languages, and operating systems, resulting in the need for a separately-implemented and entirely complete version of the remote, standalone PED system for each such targeted combination possible in the distributed computing environment
- the inefficient utilization of space as a result of the fact an individual instance of the entire standalone PED system must be associated with each remotely-resident code segment
- the potential for inefficiency and associated performance degradation in the overall system as a result of the fact that communications of debugging information throughout the distributed computing environment is usually performed at a symbolic level (involving, for example, variable names of any length).
Note that all of these drawbacks are related to the fact that the majority of the functionality of the overall PED system is concentrated in the individual, remote instances of the standalone PED system. Therefore, it was anticipated that the remediation of these drawbacks would be achieved by the remotely-extended R^n system, since the majority of the functionality of the overall system in this case would be concentrated in a single, locally-resident agent — namely, the existing EXMON execution monitor. Likewise, the remote portion of the new system would involve the minimum of functionality required for interfacing this central agent to the various individual, remotely-resident code segments. These considerations resulted in my design and implementation of the RXDM remote monitor and associated RXDM_FREND front-end interface module described in chapter 4.1

However, subsequent experimental evaluation of this remotely-extended system resulted in the realization that the overall design actually was not completely suitable for use in a conglomerate computing environment, which can involve an inhomogeneous mix of architectures, compilers, and operating systems. In other words, based on experience, it was discovered that the resulting system insufficiently dealt with certain aspects of the fundamental abstraction problem connected with such a conglomerate computing environment. In particular, although most aspects of this problem relating to architecture-dependence were handled successfully, the system nevertheless did not fully succeed in this respect since EXMON retained a need to arithmetically manipulate memory address values. Also, the system failed to deal with compiler-dependent variability in the representations of certain higher-order computational objects that are important to the debugging process, such as stack call frames.

---

1 Note that, although the design of the RXDM monitor was thus motivated by and targeted for use in the R^n programming environment, it nevertheless is suitable for general application with other systems as well — that is, at least wherever a remotely-resident mechanism for effecting execution control and debugging of remote programs running in a completely homogeneous, distributed computing environment is required.
Review of the Fundamental Abstraction Problem

Actually, one of the most significant things learned in this investigation was the nature and extent of the fundamental abstraction problem itself. The abstraction problem arises from factors related to installation-dependent differences in the configuration of conglomerate computing environments. For example, the distributed environment might consist of an inhomogeneous mix of machine architectures. Also, the various code segments residing throughout the distributed environment that constitute a given parallel/distributed program may have been generated using differing compilers, targeted either for the same or different programming languages, and even may be running under different operating systems.

In any case, the fundamental concern is how to deal with disparate representations of debugging-oriented information that can result in and from such a generally inhomogeneous situation. For example, as a result of architectural and/or compiler differences, the representations of certain 'computational objects' making up the abstract data state of a parallel/distributed program can differ from one parallel code segment to the next, perhaps in a fundamentally incompatible way, such that efficient and meaningful communication and interpretation of these objects throughout the distributed computing environment becomes a complicated issue, strongly sensitive to variations in the installation-dependent configuration of that environment. Note that these computational objects might range from instances of the 'primitive data types' such as integers, floating point numbers, address pointers, etc., to more complex and abstract program objects such as call frames on the program run-time stack. For example, with respect to the former, the corresponding representations may be incompatible due to architectural differences in regards to byte order, word size (relates to precision), sign representation, etc. For the latter type of higher-order computational objects, the corresponding representations may be incompatible due to the format or structure, and/or definition of the individual information items constituting the object, which is often dictated as much by the given compiler design (and, quite possibly, the associated, target programming language), as by any architectural considerations.

Furthermore, the abstraction problem can be compounded by any operating system differences extant in the distributed computing environment. In particular, different operating systems may provide differing mechanisms for accessing
debugging-oriented information relating to a program, as well as explicitly controlling the execution of that program. Also, different operating systems may incorporate differing interprocess communications paradigms (for example, asynchronous, signal/interrupt-driven message handling is available in UNIX, but not V), which can affect the communication of debugging-oriented information between computers in the environment. Thus, at least the design of the network interface portion of the distributed debugging system will be strongly influenced by operating system characteristics.

In order to better understand the impact of the abstraction problem in the design of a PED system, especially with regards to distributed computing environments, I developed a method for modeling any such PED scheme, that would help to reveal the strengths or weaknesses of the scheme in regards to the fundamental abstraction problem. With this method, the construction of a given model is based on the use of the basic ‘translate/debug’ stage (see section 2.3.1), which represents the relationship between two abstraction layers (or ‘abstraction environments’) containing semantically equivalent representations of the abstract data state of a program, corresponding to different levels of abstraction. Each layer is characterized by an associated ‘language’, whose constructs form the basis for expressing — at the appropriate level — abstractions within that layer. These layers are interconnected by a reciprocal pair of functions (the ‘translate/modify’ and ‘de-translate/inspect’ functions) that serve to translate relevant information across the ‘translation boundary’ separating the two levels of abstraction. For example, at the appropriate level of abstraction, these functions correspond conceptually to the translation aspect of compilation and the inverse, de-compilation aspect of debugging, respectively.

I then applied this method to various debugging systems, including the dbxtool debugging system and the remotely-extended R^n system. In particular, as a result of this analysis for the remotely-extended R^n system, I was able to ascertain that the shortcomings manifested by this system resulted not so much from defects in the design of the RXDM remote monitor, as from the original concept of simply extending the existing PED system (based on EXMON) for remote use. In other words, the PED model for the remotely-extended R^n system was inherently faulty with respect to providing an effective solution to the fundamental abstraction problem. Likewise, for the dbxtool system, the corresponding model clearly reflected the drawbacks mentioned at the beginning of this chapter. That is, although the model confirmed that the abstraction problem was solved by this particular PED scheme, this solution was
achieved in a relatively inefficient manner with respect to adaptibility, space requirements, and speed of communications.

In fact, these analyses facilitated the observation that any remote PED model is generally characterized by the following important properties:

- The manner in which the 'functionality' is distributed among the various levels of abstraction constituting the model, especially in regards to local versus remote residence in the distributed computing environment (e.g., in dbxtool, the functionality resides almost entirely on the remote side in the entity of a remote instance of dbx; in contrast, with the remotely-extended Rn system, the functionality is somewhat more evenly distributed between the locally-resident EXMON monitor and the remotely-resident RXDM monitor, although it is more heavily skewed towards the former).

- The number and types of abstraction levels involved, especially with regards to the 'languages' that characterize these levels.

- The structural complexity of the model, which determines the communications paths over which information is transferred between the abstraction environments.

**Future Work**

Since the model for the remotely-extended Rn system is crippled with respect to the fundamental abstraction problem, I proposed in chapter 5 a new model (see section 5.4.2 and Figure 5.2) for program execution and debugging in conglomerate computing environments. The model is designed to be 'idealized' in the sense that it entails features that, if properly implemented, can solve the fundamental abstraction problem, while at the same time providing a relatively efficient solution to the problems enumerated for dbxtool relating to adaptibility, space requirements, and speed of communications.

Actually, the design of the new model involved not only a revised approach to debugging, but also to compilation. In other words, during my investigation into the extent and nature of the fundamental abstraction problem, as summarized above, I realized that the design of any advanced PED system will have to recognize fully the
intimate and reciprocal relationship that characterizes the generalized debugging and translation (compilation) functions. Furthermore, the existence of this relationship occurs not only in the microscopic sense — that is, on a small, single-stage 'translate/debug' scale —, but also is reflected or expressed in the macroscopic sense, in terms of the operation of the overall system. Thus, since the relationship holds at any level of abstraction granularity, then this dictates the concrete conclusion that the design of advanced debugging features must be correlated to and integrated with a corresponding effort in compiler design, and vice-versa. This was a fundamental theme in the design of the new model.

In conclusion then, I believe that future research in the design and implementation of just such an advanced debugging system, based in particular on the 'idealized' model of Figure 5.2, would be very worthwhile. Such a system would require the development of practical, concrete realizations of the various, reciprocal 'translate/debug' functions indicated in this figure. Note that some of the relevant mechanisms already developed for the RXDM remote monitor could be used as a basis for further development in this area. In addition, this advanced debugging system would require the development of an intermediate/intercomputer transport language (IITL). The IITL would be designed to facilitate efficient and installation-independent communication of debugging-oriented information throughout the inhomogeneous, distributed computing environment, using the special debugging-oriented constructs of section 5.4.3, and abstract handles as references for all program computational objects. Also, the IITL would provide support for several advanced debugging features such as remote dynamic program restructuring. Implementation of the IITL should be made easier by using the existing R² ILOC intermediate language as a basis, to which the appropriate extensions can be made in order to achieve eventually the full functionality of the IITL.
Bibliography


[Hoo88] Robert T. Hood. Personal communication, Rice University, Houston, TX, September 1988.


Appendix A

The Frequency Principle and
Propagation of Errors in
Tree-Structured Specifications

This appendix provides a continuation of the discussion of the frequency principle, which was first presented in section 2.2.1. For convenience, I will hereby reiterate this principle: The probability of occurrence of at least a (fixed) given number of errors in a specification is related to the length of the specification, increasing as the length increases; furthermore, the overall frequency of occurrence, or density, of errors in a specification is related to the length of the specification, increasing as the length increases (the frequency principle). This principle applies primarily to the occurrence of semantics (meaning) errors in a specification; syntax errors probably will be randomly distributed throughout the specification, with a constant density with respect to the length. I believe that this principle can be largely attributed to what could be termed the 'confusion factor' — namely, as the length of a specification increases, it becomes, in general, more and more 'confusing' to the programmer, both from the standpoint of development as well as maintenance. For example, the longer the specification, the more likely it will be for the programmer to omit details, making the specification incomplete viz. the envisioned, 'correct' inputs/outputs transformation.

[ Trivial Corollary: The probability that a specification is 'correct' decreases rapidly as the length of the specification increases. Likewise, the expected number of errors in a specification increases at a more than linear rate as the length increases. ]

The following simplified example serves to illustrate this principle. Consider, in particular, a binary tree-structured specification whose interior nodes are all operators with arity 2, and whose leaves are operand objects — a good example is an abstract syntax tree (AST) with binary operators as interior nodes. Also, assume for the sake of simplicity that the binary tree is balanced and complete, and that all interior nodes are known to be correct (e.g., no errors appear in the specification of the binary operators), so that any errors in the overall specification are contained only in the leaves. Let $P_k(N)$ be the probability that, for a node at height $k \geq 0$ in the tree, the number of errors in the $2^{k+1} - 1$ leaves which are direct descendents of the
node is \( N \). Then one has the following recurrence relation for \( P_k(N) \) in terms of the inherited probabilities \( P_{k-1} \) of its two child nodes:

\[
P_k(N) = \sum_{n=0}^{N} P_{k-1}(n)P_{k-1}(N-n).
\]

The solution to this recurrence relation is heavily dependent on the nature of the internal probability distribution for errors in the objects constituting the leaves, which is largely characterized by the number of degrees of freedom in the objects. I will therefore draw upon two extreme cases, one in which the number of degrees of freedom is 1, such that any leaf node is either 'correct' or 'incorrect', and another in which the number of degrees of freedom is unbounded (at least for the sake of illustration), so that the distribution of errors in a leaf corresponds to a Poisson distribution.

**Case 1: Leaves Are Simple Objects**

The first case considers the leaves to be simple objects, such as constants or variable references, which are either correct \( (N = 0) \) or incorrect \( (N = 1) \), with the corresponding probability \( p_0 \) for the former, and \( 1-p_0 \) for the latter. Then it turns out that the above recurrence relation has as its solution the classical Binomial Distribution (see Appendix B)

\[
P_k(N) = \binom{2^k}{N} p_0^{2^k-N} (1-p_0)^N,
\]

such that \( 0 \leq N \leq 2^k \), which is normalized over \( N \) (i.e., \( \forall k \geq 0, 1 = \sum_{N=0}^{2^k} P_k(N) \)). If \( L \) is the total length — in terms of the actual number of symbols or tokens — of the (flattened) specification, then of course \( L = 2^{h+1} - 1 \), where \( h \) is the height of the tree, so that in the limit \( L \gg 1 \), this expression may be rewritten as follows:

\[
P_L(N) = \binom{L/2}{N} p_0^{(L/2)-N} (1-p_0)^N,
\]
such that $0 \leq N \leq (L/2)$. Also, the expectation value for the number of errors is

$$<N> = (1-p_0)(L/2),$$

which increases at least linearly with $L$, and even more rapidly so (at least for small $L$) if $p_0$ itself is generally decreasing with $L$ as a result of the 'confusion' factor. However, even if $p_0$ systematically decreases with increasing $L$, the expectation value will asymptotically approach straight-line behavior for large enough $L$. This behavior corresponds to the fact that the model becomes 'saturated' with errors as $p_0$ approaches zero, such that every leaf is incorrect — naturally, in this case the 'confusion' factor is irrelevant, since if the entire specification is incorrect, then things cannot get any worse!

**Case 2: Leaves Are Complex Objects**

In the next case, we assume the objects at the leaves have very high internal degrees of freedom and consequently are very rich in terms of the number of internal errors that are possible. For example, this would be the situation if the leaves were function objects, with possibly very large internal complexity (although this would be hidden from us — all we see is the function name and its arguments). In the (admittedly unrealistic) limit that the number of internal degrees of freedom in the leaves is considered unbounded (again, for the sake of illustration of extremes), then the probability distribution for $N$ errors at any leaf node can be cast in a form corresponding to the classical, normalized Poisson distribution, as follows:

$$P_0(N) = \left(\frac{\lambda_0^N}{N!}\right)e^{-\lambda_0},$$

where $\lambda_0 = -\log(P_0(0))$, and $P_0(0) = p_0$ is the probability that any given leaf node is error-free. Then the original recurrence relation has the solution (also shown in Appendix B)

$$P_k(N) = \left(\frac{(\lambda_0 2^+)^N}{N!}\right)e^{-\lambda_0 2^+},$$
such that $0 \leq N$, which will be normalized over $N$ if $\lambda_1 = \lambda_0$. Therefore, as in the previous case, this may be rewritten in terms of the length $L$ (assuming $L >> 1$), as follows:

$$P_L(N) = \left(\frac{(\lambda_0 L/2)^N}{N!}\right)e^{-\lambda_0 L/2}.$$ 

Also, the expectation value for the number of errors in this case is

$$\langle N \rangle = \lambda_0 (L/2),$$

which increases at least linearly with $L$, and even more rapidly so if $p_0$ itself decreases with $L$ as a result of the 'confusion' factor. Note that, in contrast to the first case (which displayed a 'saturation' effect since the total number of internal degrees of freedom in its leaves was bounded and finite), the expected number of errors in this case is unrestrained in terms of the order of magnitude of its variation with $L$ with different choices of forms for $p_0$ that are decreasing with $L$. This is simply because, in this case the number of internal degrees of freedom in the leaves is considered unbounded, so 'saturation' cannot occur. As a particular example, consider (say)

$$p_0(L) = \alpha e^{-\beta L}, \text{ where } \alpha, \beta > 0,$$

which results in the quadratic equation

$$\langle N \rangle = \beta L^2/2 - \log(\alpha) L / 2.$$ 

**Remarks**

Note that the expressions for $P_L(N)$ for the above two cases reflect the probability that the final result 'contains' exactly $N$ errors (all in the leaves). Consequently, the probability that there are at least $N$ errors is the same for both cases, namely
\[ P_L^< (N) = 1 - \sum_{n=0}^{N-1} P_L(n), \]

which in both cases (as can be shown) increases with increasing \( L \), and even more rapidly so if \( p_0 \) itself decreases with increasing \( L \). In addition, note that the probability there are zero (0) errors in the overall specification is also the same for both cases, namely

\[ P_L(0) = p_0^{(L/2)}, \]

which is clearly nonincreasing with increasing \( L \) (since \( p \leq 1 \)), and which becomes rapidly decreasing with \( L \) if \( p_0 \) decreases with \( L \) as a result of the 'confusion' factor.

In summary, then, these examples serve to verify the frequency principle and the attached 'trivial corollary', at least as applied to tree-structured specifications. That is, the probability that such a specification is error-free decreases rapidly with increasing length. Also, the expected number of errors in the specification generally increases at a more than linear rate with increasing length.
Appendix B

Solution to the Recurrence Relation
for Propagation of Errors in
Tree-Structured Specifications

In Appendix A, the following recurrence relation was given for expressing the probability $P_k(N)$ that, for a node at height $k \geq 0$ in the tree, the number of errors in the $2^{k+1} - 1$ leaves which are direct descendents of the node is $N$:

$$P(k,N) = \sum_{n=0}^{N} P(k-1,n)P(k-1,N-n).$$

Now, the solution to this recurrence relation is heavily dependent on the nature of the internal probability distribution for errors in the objects constituting the leaves, which is largely characterized by the number of degrees of freedom in the objects. However, the recurrence relation admits the following basic solutions (which are easily verified via inductive substitution into the recurrence relation) that are particularly applicable with respect to the two cases given in Appendix A (where $\lambda$ is some constant):

$$S_1(k,N) = \binom{2^k}{N}, \text{ such that } S_1(k,N) = 0 \text{ if } N > 2^k \text{ or } N < 0$$

$$S_2(\lambda, k, N) = \binom{\lambda 2^k}{N} \frac{N!}{N!}$$

Also, the recurrence admits the following partial solution factors (i.e., if $Q(k,N)$ is a general solution to the recurrence, and $F()$ is any partial solution factor, then $P(k,N) = F()Q(k,N)$ is also a general solution to the recurrence):

$$F_1(\lambda, k) = \lambda^{2^k} \quad \text{since substitution into the recurrence relation yields } F_1(\lambda, k-1)*F_1(\lambda, k-1) = \lambda^{2^{k-1}} = \lambda^{2^k} = F_1(\lambda, k)$$

$$F_2(\lambda, N) = \lambda^N \quad \text{since substitution into the recurrence relation yields } F_2(\lambda, n)*F_2(\lambda, N-n) = \lambda^{n+N-n} = \lambda^N = F_2(\lambda, N)$$

183
I shall now verify by substitution the correctness of the two solutions given in Appendix A.

**Case 1: Leaves Are Simple Objects**

The solution given is

\[ P(k, N) = \binom{2^k}{N} p_0^{2^k-N} (1-p_0)^N, \]

such that \(0 \leq N \leq 2^k\), which is the well-known binomial distribution function. Now, observe that, if we set \(\lambda_1 = p_0\), \(\lambda_2 = \frac{1}{p_0}\), and \(\lambda_3 = 1-p_0\), then this solution becomes

\[ P(k, N) = S_1(k, N) * F_1(\lambda_1, k) * F_2(\lambda_2, N) * F_2(\lambda_3, N) \]

such that \(0 \leq N \leq 2^k\), which by the above is in fact a general solution of the recurrence. Also, note from this that \(P(0, N) = p_0 \delta_{0N} + (1-p_0) \delta_{1N}\) (where \(\delta_{ij}\) is the Kronecker delta function), in agreement with the initial conditions for this problem.

**Case 2: Leaves Are Complex Objects**

The solution given is

\[ P(k, N) = \frac{(\lambda_1^{2^k})^N}{N!} e^{-\lambda_0 2^k}, \]

such that \(0 \leq N\). Observe that, if we set \(\lambda_2 = e^{-\lambda_0}\), then this solution becomes

\[ P(k, N) = S_2(\lambda_1, k, N) * F_1(\lambda_2, k) \]
which by the above is in fact a general solution of the recurrence. Also, note from this that, if we set $\lambda_1 = \lambda_0$, then $P(0, N) = \binom{\lambda_0^N}{N!} e^{-\lambda_0}$, which is the classical Poisson distribution, in agreement with the initial condition for this problem.