1

MicroD

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

**An Evaluation of Knowledge-Based Diagnostic Techniques
For Debugging Concurrent Software**

by

Gregory William Bond, B.Sc.

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

Department of Systems and Computer Engineering,
Carleton University,
Ottawa, Ontario, Canada

April 17, 1989

© Gregory William Bond 1989

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-51189-3

The undersigned hereby recommend to the Faculty of Graduate
Studies and Research acceptance of the thesis

*An Evaluation of Knowledge-Based Diagnostic Techniques
for Debugging Concurrent Software*

submitted by Gregory William Bond, B.Sc.
in partial fulfilment of the requirements for
the degree of Master of Engineering

A handwritten signature in cursive script, reading "Bernard Laguerre", written over a horizontal line.

Thesis Supervisor

A handwritten signature in cursive script, reading "David S. Coll", written over a horizontal line.

Chair, Department of Systems and Computer Engineering

Carleton University
April 17, 1989

Abstract

This thesis explores the possible application of knowledge-based diagnosis for locating semantic errors encountered while implementing concurrent software applications. In doing so, we propose taxonomies of diagnostic processes and diagnostic models which provide bases for a critical examination of their effectiveness in diagnostic systems. We find the behavioral diagnostic process combined with deep causal models of expectations and the faulty program as being most suitable for the diagnosis of concurrent software. We present a deep causal modelling formalism capable of representing many of the structural and behavioral characteristics of concurrent software. Comprehensive definitions of faults and errors are presented in terms of this formalism. We identify alternative sources for models of concurrent software in an implementation environment. We also assess two behavioral diagnostic techniques: one originally intended for hardware diagnosis, the other intended for diagnosing sequential Prolog programs. We determine that neither technique is especially well suited to diagnosing concurrent software but isolate those elements of both procedures which are applicable.

Acknowledgements

Many thanks to Dr. Bernie Pagurek for his enlightened approach to supervising my studies. Thanks also to the agencies which provided financial support for this research: the Natural Sciences and Engineering Research Council (NSERC) and the Telecommunications Research Institute of Ontario (TRIO). Finally, I thank Deirdre for her own unique brand of love and encouragement. She would never have let me write it alone.

Contents

1	Introduction	1
1.1	The State of Diagnosis	1
1.2	Concurrent Software Diagnosis	2
1.3	Diagnosis	3
1.3.1	Diagnosis and Debugging	3
1.3.2	Syntactic and Semantic Errors	4
1.3.3	Testing	4
1.3.4	A Diagnostic System	4
1.3.5	Knowledge-Based Diagnosis	5
1.4	Concurrent Software	6
1.4.1	Reproducible Behavior	6
1.4.2	Observability	7
1.5	Diagnosis During Software Development	7
1.5.1	Diagnosis and CASE	9
1.6	Results	9
1.7	Thesis Outline	10
2	Towards Automating Diagnosis	12
2.1	The Diagnostic Process	12
2.1.1	Structural Diagnosis	12
2.1.2	Behavioral Diagnosis	13
2.2	System Models	14
2.3	Static System Models	15
2.3.1	Structural Description	15
2.3.2	Behavioral Description	16
2.3.3	Functional Description	17
2.4	Static Causal System Models	18
2.4.1	Static Causal Shallow Models	19
2.4.2	Static Causal Deep Models	20
2.5	Static Non-Causal System Models	20
2.5.1	Static Non-Causal Shallow Models	21
2.5.2	Static Non-Causal Deep Models	22

2.6	Dynamic System Models	22
2.6.1	Dynamic Causal System Models	22
2.6.2	Dynamic Non-Causal System Models	24
2.7	Modelling to Derive Structure	24
2.8	Expectations Models for Diagnosis	26
2.8.1	Diagnosis Using Non-Causal Models	27
2.8.2	Diagnosis Using Causal Models	27
2.8.3	Shallow Models	27
2.8.4	Deep Models	28
2.8.5	Faulty Models	28
2.8.6	Fault-Free Models	28
2.8.7	Model Determinism	29
2.9	Artifact Models for Diagnosis	29
2.9.1	Observability	29
2.9.2	Reproducibility	30
2.10	Summary	31
3	A Review of Software Diagnosis	32
3.1	Cognitive Process Models	33
3.1.1	Psychological Studies	33
3.1.2	Empirical Studies	35
3.2	Debugging Tools	35
3.2.1	Fixed Autonomous Debugging Tools	36
3.2.2	Flexible Active Debugging Tools	39
3.2.3	Flexible Autonomous Debugging Tools	42
3.2.4	Flexible Passive Debugging Tools	46
3.3	A Summary of Flexible Autonomous and Passive Debugging Tools . .	49
3.4	Applicability to Our Objectives	52
4	Modelling Concurrent Software for Diagnosis	54
4.1	Extending Deep Causal Modelling	54
4.1.1	Deep Causal Models of Software Structures	59
4.2	Model Abstraction	62
4.2.1	Structural Abstraction	62
4.2.2	Behavioral Abstraction	65
4.2.3	Object Abstraction	66
4.2.4	Detail Abstraction	67
4.3	Determinism	67
4.4	Dynamic Models	67
4.5	Modelling Faults and Errors	68
4.5.1	Faults	71
4.5.2	Errors	72

4.6	Comparable Software Models	74
4.6.1	State-Based Models	74
4.6.2	Program Dependence Models	76
4.7	Summary	78
5	Concurrent Software Modelling Sources	79
5.1	The Environment	79
5.2	Static Causal Artifact Models	81
5.3	Dynamic Causal Artifact Models	81
5.3.1	Artifact Reproducibility	81
5.3.2	Artifact Observability	82
5.3.3	Hardware Monitoring	82
5.3.4	Other Issues	83
5.3.5	Alternatives	84
5.4	Expectations Models	87
5.5	Summary	87
6	Diagnostic Processes for Causal Models	89
6.1	Violated Expectations	89
6.1.1	Expectations Models	90
6.1.2	Artifact Models	90
6.1.3	The Diagnostic Process	90
6.1.4	Evaluation Mechanisms	91
6.1.5	Testing and Measurements	93
6.1.6	Fault Coverage	95
6.1.7	Assumptions	97
6.2	PDS6	97
6.2.1	Artifact Models	98
6.2.2	Expectations Models	99
6.2.3	Termination with Incorrect Output	100
6.2.4	Termination With Missing Output	102
6.2.5	Nontermination Error	103
6.2.6	Incorrect Solution vs. Missing Solution	104
6.3	Applicability to Our Objectives	105
6.3.1	Nontermination	105
6.3.2	Nondeterminism	105
6.3.3	Intrastructural Dependency Constraints	106
6.3.4	Refining the Search Space	106
6.3.5	Model Detail	106
6.3.6	Fault Coverage	107
6.4	Summary	107

7	Conclusions and Future Research	108
7.1	A Research Summary	108
7.1.1	Diagnostic Taxonomies	108
7.1.2	Deep Causal Modelling	109
7.1.3	Model Sources	109
7.1.4	Diagnostic Processes	109
7.2	Assessment of Results	110
7.3	Future Research	110
7.3.1	Diagnostic System	111
7.3.2	External Environment	111
7.3.3	Expectations Model Manager	112
7.3.4	Artifact Model Manager	112
7.3.5	User Interface	112
7.3.6	Diagnosis Manager	112
	References	114
A	Reinterpretation of Shapiro's Diagnostic Procedures	122
A.1	Incorrect Solution as a Missing Solution	122
A.2	Missing Solution as an Incorrect Solution	126

List of Figures

1.1	A fault tolerant system	3
1.2	A diagnostic system	5
1.3	Phases of Software Development	7
1.4	Diagnosis models for faulty development phases	8
2.1	Structural diagnosis	13
2.2	Behavioral diagnosis	14
2.3	A taxonomy of diagnostic models	14
2.4	Structural description of a first-order filter	16
2.5	AND/OR gate schematic	18
2.6	Causal shallow model of AND/OR gate configuration	19
2.7	Causal deep model of the AND/OR gate configuration	21
2.8	Non-causal shallow model of AND/OR gate configuration	22
2.9	Schematic of a 2-bit sequential multiplier artifact	23
2.10	Causal deep model of a 2-bit sequential multiplier	24
2.11	Dynamic causal model of a 2-bit sequential multiplier	25
3.1	Gould's gross descriptive model of debugging	34
3.2	A program slice on the variable Z	34
3.3	A Summary of Flexible Autonomous Debugging Tools	50
3.4	A Summary of Flexible Passive Debugging Tools	51
4.1	Deep causal model of a 2 bit adder	55
4.2	Sequential program and its deep causal model	56
4.3	Data variable representation	57
4.4	Deep causal models of individual statements	58
4.5	Variable assignment model	59
4.6	If-Then-Else model	60
4.7	Accept entry Model	61
4.8	Call entry Model	62
4.9	Selective accept Model	63
4.10	Block abstraction models	64
4.11	Modelling data variable abstraction	65
4.12	Procedure call model	66

4.13	Structural abstraction of three synchronizing tasks	67
4.14	Artifact and static deep model of three communicating tasks	69
4.15	Dynamic model of three communicating tasks	70
4.16	Deep causal models of two program fragments	71
4.17	Dynamic models of two program fragments	72
4.18	Dynamic models of two program fragments	73
4.19	An example state transition diagram	75
4.20	A state transition diagram that cannot be abstracted	76
4.21	A program flow graph and a deep model of the same code section . . .	77
5.1	Current development environment hardware configuration	80
5.2	An event monitor for a distributed system	83
5.3	A comparison of dynamic artifact model sources	88
6.1	The violated expectations process	92
6.2	Half-split measurement	94
6.3	Short circuit artifact and its model	96
6.4	Exhaustive evaluation of expected behavior	96
6.5	A Prolog program and a possible proof tree	99
7.1	High level structural decomposition of the diagnostic system	111
A.1	Static Causal Artifact Model	123
A.2	Static Causal Model of Expectations	123
A.3	Dynamic Causal Artifact Model	123
A.4	Dynamic Causal Model of Expectations	124
A.5	Dialogue for Incorrect Solution	124
A.6	Dialogue for Missing Solution	125
A.7	Static Causal Artifact Model	126
A.8	Dynamic Causal Model of Expectations	126
A.9	Dialogue for Missing Solution	127
A.10	Dialogue for Incorrect Solution	128

Chapter 1

Introduction

1.1 The State of Diagnosis

Diagnosis is best described as an informal process. Competency in diagnosis is not achieved by learning a formal technique but is honed by exposure to situations where diagnosis is required. More comparable to an art than a science, diagnosis has resisted attempts to be formalized. In the field of software development, diagnosis in the form of debugging plays a large role in the design and maintenance phases of a software system.

An associated impediment to the efficient diagnosis of a system is its complexity. Systems diagnosis normally proceeds in an ad-hoc fashion by interpreting knowledge gleaned from past diagnostic experiences, knowledge of a system's internal elements and knowledge of a fault's symptoms. The more complex the system, the more involved the interpretation process becomes.

A poignant example of inefficient diagnosis in the face of diagnostic information overload occurred during the accident at Three Mile Island nuclear power plant in 1979. The accident triggered at least 100 annunciators in the control room within the first few minutes of its occurrence [Nelson82]. The control room operators were unprepared for the interpretation of so many diagnostic signs within the time constraints demanded of the reactor failure.

The lack of formalism in systems diagnosis manifests itself not only as being difficult to perform efficiently but also being difficult to teach. This has been a

cause of concern in the medical field [Fischhoff79] :

Increasingly, medical educators are realizing that for their training to be effective, it must be embedded in a theory of how medical students and expert physicians perform diagnosis.

The situation is similar in all fields where systems diagnosis is practiced: the lack of a formal understanding of diagnosis has led to widely varying and largely incomprehensible techniques that are put into practice by a small number of expert diagnosticians. Their expertise is normally gained by repeated exposure to diagnostic situations and, in the case of man-made systems, by their role in the implementation of the system being diagnosed.

1.2 Concurrent Software Diagnosis

Concurrent software programs are often employed as subsystems of embedded systems in application areas including avionics, telecommunications and process control. The development of concurrent software systems currently suffers from all the problems associated with the diagnosis of complex systems. There is no complete formal methodology for finding faults in this domain. The domain is rife with diagnostic information sources. Competent diagnosis is normally restricted to those individuals that design the software.

A primary goal of this thesis is to evaluate the applicability of knowledge-based diagnostic techniques for debugging concurrent software. We would like to discover what knowledge-based diagnostic techniques offer the diagnostician over traditional debugging techniques. Another goal of this thesis is to identify a diagnostic technique with provable properties such as fault coverage. We would also like to investigate what form a knowledge-based system for concurrent software might take with the implementation of a prototype system in mind.

We restrict our problem domain to concurrent software applications targeted for tightly-coupled multiprocessor execution environments. The diagnostic technique should be able to locate faults manifesting semantic errors during the implementation phase of software development. We also investigate what computer aided

Figure 1.1: A fault tolerant system

software engineering (CASE) has to offer the diagnostician. The rest of this chapter is devoted to overviews of our terminology, our research objectives and our results. This is followed by a brief outline of the thesis contents.

1.3 Diagnosis

Diagnosis is the act of locating a fault in a system. We refer to the system being diagnosed as the *artifact*. Diagnosis differs from *error detection* and *fault correction*. An *error* is the unexpected behavior observed which indicates the presence of faults in an artifact. A *fault* is a discrepancy between actual and expected artifact structure which manifests errors in the artifact.

Put in context, combining error detection, fault location and fault correction subsystems with an artifact forms an autonomous fault tolerant system [Hudlicka86] as shown in Figure 1.1.

1.3.1 Diagnosis and Debugging

Debugging consists of detecting a software error, locating the associated fault and then correcting the fault. However, this thesis is concerned only with the location of a fault given that an error has been detected - the diagnosis part of the debugging process.

The common definition of a *bug* encompasses both the error observed in a program and the fault itself. As this definition is imprecise, we shall continue to speak in terms of software errors and software faults.

1.3.2 Syntactic and Semantic Errors

Programming languages have associated with them particular *syntax* and *semantics*. Syntax refers to the rules that describe what strings of characters from the language's alphabet are valid programs. The semantics of a program language are rules which describe the *meaning* of a program [Aho86].

Language compilers are quite adept at detecting syntax errors and locating their associated faults but are limited to detecting only a few semantic errors. This thesis examines the location of faults associated with semantic errors.

1.3.3 Testing

Testing can be an integral part of the diagnostic process. However, some confusion can arise as an alternative form of testing is employed at different stages of the development lifecycle. Testing can play a role in both the processes of fault detection and fault location. However, testing to detect faults is a different process from testing to locate faults. Thus, it is the goal of the test which categorizes the testing process employed.

Testing for fault detection is the process more commonly referred to as 'system verification.' System verification techniques can detect a fault but do not necessarily locate a fault.

Testing for fault location or 'diagnostic testing' consists of choosing inputs based on knowledge of suspected faulty system elements. Satisfying a diagnostic test results in the elimination of some elements from the set of potentially faulty elements and, ultimately, the identification (location) of any faulty elements.

1.3.4 A Diagnostic System

Figure 1.2 illustrates the diagnostic subsystem of Figure 1.1 considered alone. It consists of an *artifact model*, an *expectations model* and the *diagnostic process*.

Figure 1.2: A diagnostic system

Ideally, the artifact model is some representation of the artifact's *actual* structure and behavior. The expectations model is some representation of the artifact's *expected* structure and behavior. Expectations models can take many forms. They may be maintained in the diagnostician's head or as design specifications on paper or on a computer.

The diagnostic process compares the two models to locate faults. Note that decoupling the diagnostic process from the models allows the exchange of models which represent different artifacts and expectations without having to alter the diagnostic process.

1.3.5 Knowledge-Based Diagnosis

A knowledge-based diagnostic system is a diagnostic system where the artifact and expectations models constitute a knowledge base and the diagnostic process is an inference procedure.

Given complete models of expectations and the artifact, fault location is trivial – locate all structural discrepancies between the artifact model and the expectations model. However, the availability of complete models is rare. For example, the actual structure and behavior of the artifact may not be wholly observable permitting the construction of only a partial model of the artifact's structure and behavior. Similarly, the artifact's expected structure as described by the expectations model may be known but, given a complex artifact, enumerating all possible expected behaviors may require too much overhead to warrant their inclusion in the model.

In these situations, the diagnostic process must do more than merely compare

the two models, it must *infer* the missing elements of the model before it can compare them. This may take the form of synthesizing a complete model of the artifact's actual internal structure and behavior given only a partial description or simulating the artifact's expected behavior from a description of its expected structure. This ability to infer missing details in incomplete models makes a knowledge-based diagnostic process attractive.

1.4 Concurrent Software

Concurrent software for embedded systems¹ can be characterized by:

- Concurrent execution of *tasks*² – Concurrency can be *virtual* when concurrent software executes in a uniprocessor environment or *real* when it executes in a multiprocessor environment.
- Intertask communication/synchronization³ – Intertask communication mechanisms include the Ada rendezvous, semaphores, monitors, mailboxes *etc.*
- Preemptive task scheduling – This is a characteristic associated with, among other things, the execution of interrupt subroutines (ISR's).

Our intention is to investigate fault location in concurrent software executing in a tightly-coupled multiprocessor environment. Specifically, up to 16 Motorola 68020-based DY-4 DVME-134 single board computers [DVME-134] communicating across a VME bus. The concurrent software which executes on the system is Unison [Unison87a]. However, in this thesis we refer to the Ada programming language [ALRM83]. Ada has similar semantics to Unison's and Ada has the added virtue of being more widely known.

1.4.1 Reproducible Behavior

In the execution environment we are faced with a number of diagnostic difficulties caused by non-reproducible behavior:

¹Sometimes referred to as *realtime software*.

²The term *tasks* refers to coarse grain concurrent program entities sometimes referred to as *processes*.

³This is sometimes referred to as *interprocess communication* (IPC).

Figure 1.3: Phases of Software Development

- the processors of the multicomputer are not synchronized;
- the intertask communications mechanism provided by the runtime system of the concurrent language is nondeterministic;
- an external environment consisting of non-periodic input signals driving ISR's.

Of course non-reproducibility is not a problem when the system is behaving correctly. Only when it behaves incorrectly does the non-reproducible behavior manifest itself as *intermittent errors*.

1.4.2 Observability

Given a suitable symbolic debugger, the software artifact is observable to low levels of structural and behavioral detail. This is a valuable source of information when constructing an artifact model. However, this level of observability is not gained without a cost. By inserting the probes necessary for gathering this information one is, in fact, altering the structure and behavior of the artifact. In doing so a previously observed error might disappear or new errors may appear. Faults located with the probes in place will only be applicable to the structure of the artifact with the probes in place. Once the probes are removed, the structure of the artifact changes and new faults may be present.

1.5 Diagnosis During Software Development

The traditional model of software development describes four phases: requirements analysis, design, implementation and maintenance. As shown in Figure 1.3 the output of each phase serves as the input for the next.

Phase	Artifact	Expectations Model
Requirements	Requirements Specification	Problem Statement
Design	Design Specification	Requirements Specification
Implementation	Code	Design Specification
Maintenance	Installed Code	Code

Figure 1.4: Diagnosis models for faulty development phases

From a diagnostic perspective, the input serves as an expectations model for the output artifact. For example, one's expectations concerning the code developed during the implementation phase are encapsulated in the design specification.

There are two potential reasons for a faulty output artifact: the transformation process from input to output may be faulty or the input expectations model may be faulty indicating a fault in a preceding transformation process. In the lexicon of software engineering we hear of *design errors* and *implementation errors*. These are errors caused by a fault in the corresponding transformation process. Isolating the phase of the lifecycle which has introduced an error identifies the models required for diagnosis. Figure 1.4 illustrates the applicable models for diagnosing faults introduced at different phases of software development.

If operational specifications (for rapid prototyping) are employed, the developer may assess system behavior at phases preceding the implementation phase. Observing behavior earlier in the development process allows for earlier detection of errors. Upon detection of an error a knowledge-based diagnostic system might be employed to locate faults in a requirements or design specification.

One should note, however, that the expectations models are less detailed than the artifact being diagnosed. This imbalance in detail affects the extent to which faults can be localized since faults can only be localized to the level of detail of the least detailed model.

Much effort is being directed to automating the transformations of input to output. A desirable property of the transformation is preserving behavioral correctness. At this point in time correctness preserving transformations are not a reality. However, it is interesting to consider what impact their use would have on the diagnostic process.

The implication of diagnosing an error detected in an artifact produced using correctness preserving transformations is that either the problem statement must be faulty or a transformation has been implemented incorrectly. In the former case the problem statement must be reformulated. In the latter case the transformation itself must be diagnosed. Note that in using correctness preserving transformations the input to the transformation can no longer serve as an expectations model of the output artifact. It is effectively an abstract model of the output artifact.

This thesis concentrates its effort towards locating faults introduced by semi-automatic transformations during the implementation phase. By semi-automatic transformations we mean that *skeleton code* is automatically generated from the design specification but the accompanying *flesh code* must be manually generated.

1.5.1 Diagnosis and CASE

CASE provides software developers with an integrated set of tools which support some or all phases of software development [Rich88, Buhr88, Statemate87]. A natural addition to a CASE environment would be a knowledge-based diagnosis tool. Since the model sources outlined in §1.5 would be in computer accessible format the diagnosis system would have direct access to them.

1.6 Results

This thesis examines two related fields of research: knowledge-based diagnosis and debugging concurrent software. The result is a unification of the two fields which clearly defines the characteristics associated with the problem of knowledge-based diagnosis of concurrent software.

We propose terminology and conceptual taxonomies which draw explicit links between the two fields. In particular, we propose classifications of the diagnostic

process and diagnostic models which form the bases for a critical investigation of their suitability for software diagnosis. The classifications also provide common grounds for comparison of a number of software diagnosis systems and software diagnosis tools.

The most applicable diagnostic model for concurrent software is found to be lacking in its representational abilities. To overcome this problem we propose extensions to the traditional diagnostic modelling formalism to accommodate the unique characteristics of concurrent software programs. We find that, in fact, the extended formalism is more general in its representational abilities than the traditional formalism. In terms of this formalism we propose comprehensive classifications of faults and errors which can occur during concurrent software execution.

From the perspective of sources for the software models, we propose four alternatives which make use of many of the tools in the existing software implementation environment and require a number of additional sophisticated tools. The four alternatives are found to complement phases of concurrent software implementation encountered in the implementation environment.

For the diagnostic process chosen as most applicable for the concurrent software domain we present a detailed evaluation of its existing implementations. Doing so reveals some previously undiscovered parallels between them. We conclude that no existing process is particularly well suited to the task of concurrent software diagnosis but we fall short of specifying a suitable technique based on our findings.

The results of this thesis clearly define the problems associated with the diagnosis of concurrent software and set the stage for a formalized knowledge-based diagnostic technique.

1.7 Thesis Outline

Chapter 2 proposes a taxonomy of diagnostic processes and models which clarifies the relationship between the process and its models. In Chapter 3 the same taxonomy is imposed on a number of debugging tools in an effort to highlight their diagnostic capabilities. In light of the findings of the preceding chapters we make a preliminary decision regarding the diagnostic process and model best suited to

the task of concurrent software diagnosis. Chapter 4 proposes a modelling formalism which accommodates a number of representational difficulties encountered when modelling concurrent software for diagnosis. In this chapter we also propose fault and error classifications in terms of the proposed formalism and examine examples from the concurrent software domain. Chapter 5 investigates possible modelling sources in the software implementation environment. Four alternative model sources are proposed which integrate with the phases of concurrent software implementation encountered in our particular environment. Chapter 6 investigates diagnostic processes of the type chosen for diagnosing concurrent software in Chapter 3. We conclude that no existing technique is well suited to the task and put forward suggestions for their improvement. Chapter 7 discusses future directions for our research and reviews the results of this thesis. We conclude by presenting the preliminary structural configuration of a knowledge-based system for the diagnosis of concurrent software executing in a multiprocessor environment.

Chapter 2

Towards Automating Diagnosis

Before considering the specifics of concurrent software diagnosis the terminology associated with the area of automated diagnosis must be established. In this chapter we propose taxonomies of diagnostic processes and models. The taxonomies aim to clarify important concepts encountered in this rapidly evolving field and will be used as a basis for discussion in ensuing chapters. In the final sections of this chapter, we discuss attributes of diagnostic models and examine their effects on diagnostic systems which employ them.

2.1 The Diagnostic Process

Chapter 1 defined diagnosis in terms of locating structural inconsistencies between models of expectations and reality. Depending on the models available diagnosis can proceed either as a *structural diagnosis* or a *behavioral diagnosis*.

2.1.1 Structural Diagnosis

Figure 2.1 illustrates the process of structural diagnosis. Given models describing the artifact's expected and actual structure the process proceeds to locate faults by comparing the two models.

An example of structural diagnosis would be the 'structural' examination a physician employs when diagnosing a patient with a compound fracture. The location

Figure 2.1: Structural diagnosis

of the fracture can be derived without any knowledge of patient symptoms by performing a visual inspection of the patient.

This form of diagnosis is best suited to artifacts with easily observable structure. Any information regarding how the fault manifests itself as symptoms is ignored. Thus, some other source of information must be chosen to efficiently guide the model comparison process (perhaps the probability of failure of each structure) or the process is reduced to an exhaustive search.

There can be no guarantee that repairing a fault located using this technique will eliminate any symptoms observed in the faulty system unless the process is exhaustive *i.e.* is guaranteed to detect all the faults in a system.

2.1.2 Behavioral Diagnosis

A behavioral diagnosis locates system faults given a fault's symptoms as shown in Figure 2.2. This process infers the location of a fault without having to explicitly compare structure. An example of behavioral diagnosis is the diagnostic technique a physician employs which begins with asking what is bothering the patient. The physician then uses knowledge of the symptoms to guide the diagnosis.

This form of diagnosis is best suited to artifacts for which structure is difficult or impossible to examine but behavior is more readily observable. It is also efficient in locating faults since it uses information regarding faulty system behavior to help localize potential faults.

Figure 2.2: Behavioral diagnosis

Figure 2.3: A taxonomy of diagnostic models

2.2 System Models

In the following sections we present a taxonomy of system models for diagnosis. As illustrated in Figure 2.3 system models may be classified as *static*, describing the passive system or *dynamic*, describing the active system. When submitted to an *evaluation mechanism* a static model is converted to a dynamic model. The mechanism essentially simulates the activities of the active system from a description of the static system.

For example, the static system model may describe system structure and its general relationship with system behavior. When submitted to an evaluation mechanism with suitable initial behavioral conditions, a dynamic model is generated describing the resultant behavior of the dynamic system.

Static and dynamic models may further be classified as *causal* or *non causal* reflecting the organization of a model's description. In order to better understand systems we sometimes impose a notion of *causality* upon a model in order to explain the dynamic system. Such a model is referred to as a *causal model*. A causal model contains information regarding the causes of *events* in a system. From a causal viewpoint, events are said to cause other events. From a temporal viewpoint, a given event can only occur *after* the event which caused it.

Static models are further classified as either *shallow* or *deep*. The meaning of the classifications, originally coined by [Davis82b] and [Michie81], have been altered somewhat here. The term deep model originally referred to what we refer to here as a static causal deep model. The term shallow model originally referred to what we refer to here as a static non-causal shallow model. Thus, the terms originally classified the causality of the model. We use the term to loosely classify the descriptive completeness of the model.

We leave the details of these classifications to the following discussion.

2.3 Static System Models

Since diagnosis involves isolating structural discrepancies the diagnostic models must describe structure. Diagnosis exploiting the knowledge of an error's presence requires that the model describe behavior as well.

We first investigate a general static system model incorporating both structure and behavior as *structural* and *behavioral descriptions*. Then we shall describe variations of this model for the purposes of diagnosis. The emphasis here is on describing static system models for evaluating *behavior* in the dynamic model. In a later section (§2.7) we examine analogous models for evaluating *structure* in the dynamic model.

2.3.1 Structural Description

Our understanding of system structure comes from observations of the static system which is comprised of a number of inter-connected *structures*. Associated with the structures of the system are *behavioral variables*. The behavioral variables are

Figure 2.4: Structural description of a first-order filter

associated with measurable attributes of structures where system behavior is defined to be the values of behavioral variables as they vary with time in the dynamic system. Note that a static system cannot exhibit behavior.

Structures are linked via *connections* between behavioral variables of unique system structures. Connections are known either from the design of the system or inferred by observation of the dynamic system. A *structural description* of a system is the set of structures which comprise the system, the set of connections between those structures and the behavioral variables associated with those structures.

An example of a simple system graphically defined in these terms is the schematic of a first-order filter shown in Figure 2.4. The filter consists of two structures: resistor R and capacitor C . Behavioral variables of the system might be the voltage across R , V_R and the voltage across C , V_C .

2.3.2 Behavioral Description

In many cases explicitly enumerating the possible behaviors of a system is infeasible. The alternative is to provide the static system model with the information required for evaluating system behavior when submitted to an evaluation mechanism.

Connections between variables can be imposed with directionality rendering an *interstructural dependency* between variables. Dependencies between behavioral variables can be *unidirectional* or *bidirectional*. A unidirectional dependency between two variables implies that the value of one of the variables (the dependent

variable) is dependent on the value of the other (the independent variable) and not vice versa. An example of unidirectional dependence is found in an AND gate. The gate's output value is dependent on its input values but its input values are not normally dependent on its output value. A bidirectional dependency between two variables implies that their values are mutually dependent. An example of bidirectionally dependent variables are the resistor voltage V_R and capacitor voltage V_C of Figure 2.4.

Apart from interstructural dependencies, *intrastructural dependencies* are assumed to exist between all the behavioral variables of a structure. A variable at a given structure may be dependent on many other variables at that structure. For example, the output variable of a two input AND-gate is dependent on the values of both the gate's input variables.

Behavioral variables can be quantitatively or qualitatively¹ evaluable. The evaluation of behavioral variables requires a suitable formalism for constraining the dependencies between variables and an evaluation mechanism to accomplish the evaluation. A *behavioral description* of a system is the set of all constrained dependencies between behavioral variables of the system. Note that the constrained dependencies will indicate the directionality of the dependencies. A behavioral description of the first order filter might be $V_o(t) = \frac{1}{C} \int_{-\infty}^t i dt + iR$. The differential equations act as constraints on the bidirectional variable dependencies.

2.3.3 Functional Description

The *function* of a system is often equated with the behavior of a system. For the purposes of this thesis, function will be defined as "the purpose of a structural element or connection in producing the behavior of the system" [Kuipers84]. A *functional description* of a system conveys the overall operation of a system without drawing on detailed behavioral and structural descriptions. For example, the function of a first-order filter could be to attenuate signal components occurring at high frequencies.

¹A qualitative evaluation consists of assigning to the behavioral variables of a system a zero, positive or negative value. For a thorough overview of the area, refer to [Bobrow85]

Figure 2.5: AND/OR gate schematic

2.4 Static Causal System Models

Static causal models are static models whose descriptions are organized to reflect the causal relationships between system structure and behavior. A static causal model submitted to an evaluation mechanism with initial behavioral conditions results in a dynamic causal model of system behavior. The behavior described by the causal model may be understood in terms of the *local* interactions of behavior between structures comprising the system thereby providing a better understanding of the processes at work responsible for the behavior.

For example, consider the AND/OR gate schematic shown in Figure 2.5. When describing the potential behavior of the gate configuration, a causal description might state that the AND gate will output a value O_1 in accordance with the values of its inputs I_1 and I_2 and its device truth table. The OR gate will subsequently output a value O_2 in accordance with the values of the AND gate's output O_1 , input I_3 and its own device truth table. The reference to the AND gate's output *directly* affecting the behavior of the connected OR gate constitutes a description in terms of local behavioral interactions. A description of behavior in these terms is necessarily causal. In our description it is implied that the OR gate's output is *caused* by the AND gate's output and I_3 .

The static causal model describes the locality relationships via dependency constraints. Values of interdependent variables between structures are constrained to be equal. So constrained, interstructural dependencies effectively serve as interfaces between structures. For example, in Figure 2.5 we consider the AND gate's output O_1 to be equivalent to the OR gate's input. Intrastructural dependencies are constrained to reflect the behavioral constraints between variables local to the structure. For example, in Figure 2.5 we could state the relationships between input

AND Rules	OR Rules
R_1 : if $I_1 = 0$ and $I_2 = 0$ then $O_1 = 0$	R_3 : if $O_1 = 0$ and $I_3 = 0$ then $O_2 = 0$
R_2 : if $I_1 = 0$ and $I_2 = 1$ then $O_1 = 0$	R_4 : if $O_1 = 0$ and $I_3 = 1$ then $O_2 = 1$

Figure 2.6: Causal shallow model of AND/OR gate configuration

and output variables of a gate as a truth table. Organized this way, an evaluation mechanism evaluates behavior, structure by structure, along a causal path in much the same way we believe the behavior to be manifested in reality.

2.4.1 Static Causal Shallow Models

Static causal shallow models² are comprised of rules *relating* behavioral values to the structures they occur at. The rules are incomplete in that they don't provide an exhaustive description of the behaviors possible in a system. Shallow models expressed as rules combine behavioral and structural descriptions for particular occurrences of behavior. The evaluation mechanism (an inference procedure) does not determine the values of behavioral variables but determines which of the given rules apply for a particular system behavior. The order of rule use in the deduction chain indicates the causality of behavioral interactions in the modelled system. For example, an incomplete description of the behavior of the AND/OR gate configuration (Figure 2.5) is shown in Figure 2.6. Given input values $I_1 = 0, I_2 = 1, I_3 = 0$ the behavior generation mechanism uses rules R_2 and R_3 implying $O_1 = 0$ and $O_2 = 0$. In this example rule R_2 would fire before R_3 indicating that the behavior of the OR gate is, in part, caused by the AND gate. Causal shallow models were first employed in medical diagnosis systems such as CASNET [Weiss78] and ABEL [Patil81].

²Since the shallow and deep classifications are applicable to static models and not to dynamic models we will sometimes refer to shallow and deep models without including the static qualification.

2.4.2 Static Causal Deep Models

Static causal deep models consist of complete structural and behavioral descriptions. The evaluation mechanism simulates the processes known to generate the behavior in the real world. A much wider range of behavior can be generated from deep models than shallow models. For the AND/OR gate schematic of Figure 2.5, a graphical depiction of its deep model is shown in Figure 2.7³. Apart from the generic representation of system structures as rectangles and the indication of interstructural dependency directionality, the schematic and the deep causal model are identical⁴.

Sussman and Steele [Sussman80] use a constraint language for deep modelling of electrical networks. Using a method called *constraint propagation*, they are able to evaluate quantitative behavior of the network. Methods derived by de Kleer [deKleer77, deKleer79], Forbus [Forbus81, Forbus84] and Hayes [Hayes79] utilize similar techniques for the causal evaluation of qualitative behavior.

2.5 Static Non-Causal System Models

In comparison to static causal models, static non-causal models have no locality requirements for dependency constraints. The evaluation mechanism therefore produces a dynamic model whose behavior cannot be understood in terms of causal interactions of behavior between connected structures. Causality may, in fact, be imposed on the behavior of the dynamic model but the causality is not necessarily related to structure which limits the model's effectiveness for diagnosis.

³This thesis frequently depicts deep causal models graphically. The graphical representation accurately conveys the structural description of a system and the interstructural dependencies of a system. However, the graphical representation cannot convey the constrained intrastructural dependencies. We will refer to these illustrations as deep causal models throughout the remainder of this thesis. Tables that explicitly describe intrastructural dependencies will be included when the dependencies are not obvious.

⁴In fact, the schematic graphically conveys more information concerning how intrastructural dependencies are constrained by utilizing differently shaped structures to represent differently constrained dependencies. Of course, this technique is no longer useful when the number of differently constrained structures becomes too numerous to remember.

Constrained Intrastructural Dependencies					
AND Structure			OR Structure		
I_1	I_2	O_2	O_1	I_3	O_2
0	0	0	0	0	0
0	1	0	0	1	1
1	0	0	1	0	1
1	1	1	1	1	1

Figure 2.7: Causal deep model of the AND/OR gate configuration

2.5.1 Static Non-Causal Shallow Models

A static shallow model in a non-causal context is a set of rules which relate arbitrary structural and/or behavioral attributes of a system. An inference procedure applied to the rule base can generate a limited number of dynamic models describing system behavior. For the AND/OR gate configuration of Figure 2.5, a static non-causal shallow model might be that shown in Figure 2.8. Note that the rules from which the behavior can be inferred are not necessarily associated with any unique structure as they were in §2.4.1. Given input values $I_1 = 1, I_2 = 0$ and $I_3 = 1$ implies that $O_2 = 1$ and $O_1 = 0$ using rules R_3 and R_2 . In this example, rule R_3 fires before R_1 , but this does not indicate anything about the modelled system's causality since the rules are not associated with unique structures. This type of model was used in first generation knowledge-based diagnostic systems such as MYCIN [Shortcliffe76].

Rules
R_1 : if $I_1 = 0$ and $O_2 = 0$ then $I_3 = 0$
R_2 : if $O_2 = 1$ and $I_2 = 0$ then $O_1 = 0$
R_3 : if $I_1 = 1$ and $I_3 = 1$ then $O_2 = 1$

Figure 2.8: Non-causal shallow model of AND/OR gate configuration

2.5.2 Static Non-Causal Deep Models

A static deep model in a non-causal context implies a model description that is able to produce a wide range of behaviors when submitted to an evaluation mechanism just as the causal deep model does. However, behavior evaluated using non-causal deep models cannot be described with a causal interpretation.

The common form of non-causal deep models are differential equations which provide constraints on the dynamics of behavioral variables. An analytic technique is utilized to evaluate quantitative behavior and a similar technique proposed by Kuipers [Kuipers84] uses a weaker form of the differential equation to evaluate qualitative behavior. Behavior evaluated in this way is explained in terms of the solution of a set of differential equations with no reference to system structure or causality.

2.6 Dynamic System Models

Dynamic models are constructed from observing the dynamic system or are a product of submitting a static system model with initial conditions to an evaluation mechanism. The dynamic model describes the values of behavioral variables of a system as they vary with time.

2.6.1 Dynamic Causal System Models

A dynamic causal model reflects the causality of system behavior. If the model is a result of evaluating a static causal model then any bidirectional dependencies will have a unidirectional causality imposed upon them. de Kleer [deKleer79] refers to

Figure 2.9: Schematic of a 2-bit sequential multiplier artifact.

this as deriving a *causal topology* from a *device topology*. For static models with only unidirectional dependencies the causality has already been imposed. This identification of causality is analogous to identifying the input and output variables of a system. Input and output variables can be determined by examining the directionality of interstructural dependencies. At a given structure the behavioral variables dependent on the values of variables at connected structures are input variables; the variables which influence the values of variables at connected structures are output variables.

Figure 2.9 shows a schematic diagram of a 2-bit sequential multiplier with a 4-bit accumulator⁵. Figure 2.10 shows its corresponding causal deep model⁶. Figure 2.11 illustrates the evaluated behavioral variables in a dynamic causal model of the multiplier.

A static causal deep model is essentially a representation of all possible behavioral interactions of a system. The dynamic causal model represents one such causal interaction. Thus, for deep models with unidirectional dependencies and no feedback loops⁷ the static deep model and the dynamic model are identical save for the

⁵With each clock pulse the multiplier shifts the contents of the A register down 1 bit and shifts the contents of the B register up one bit. A high input enables the driver - a low input disables it.

⁶The use of this circuit as an example has been borrowed, with modifications, from [Hamscher84].

⁷A feedback loop is indicated by a unidirectional dependency path by which a behavioral variable x influences another variable that is an antecedent in the dependency path leading to x [Iwasaki86].

Figure 2.10: Causal deep model of a 2-bit sequential multiplier

evaluation of behavioral variables. When a feedback loop is present, the dynamic model can contain multiple instantiations of variables, structures and dependencies as shown in Figure 2.11.

2.6.2 Dynamic Non-Causal System Models

A dynamic non-causal model describes the values of system variables as they vary in time but conveys no information regarding the causality of behavior manifested in the system.

Iwasaki and Simon [Iwasaki86] describe a method for deriving a causal ordering of behavioral variables from differential equations – a static non-causal deep modelling formalism. The causal ordering derived can be used as the basis for explaining the behavior generated from the differential equations using analytic techniques but as the static model is non-causal, the causality is not explained in terms of structure.

2.7 Modelling to Derive Structure

The previous sections have dealt with evaluating system behavior from static models describing structure and behavior. This section examines the less common techniques for evaluating system *structure* from an alternate form of static model. The system being modelled here is one that, when active, configures primitive structural elements to form a system with some desired characteristics. An example of such a

Figure 2.11: Dynamic causal model of a 2-bit sequential multiplier

system is a programmer, cognizant of primitive program structures, arranging the structures in such a way as to construct a program which achieves some desired behavior. The static programmer system might be modelled as a set of primitive program structures organized according to how they might be interconnected. When submitted to an evaluation mechanism, the interaction of the evaluation mechanism with the static model simulates the creative process a programmer utilizes when constructing a program. Initial conditions for the evaluation mechanism might be the desired behavior of the program to be constructed. The resultant dynamic model is a program achieving the desired behavior which consists of primitive program structures connected together according to their rules of organization. The concept of an active system organizing structures to form a complete system is analogous to the active system which exhibits behavior discussed in the preceding sections.

The analogy extends to the concepts of causality. Where structural locality can be used as an organizational principle of static causal models for evaluating behavior, a system implementor's *intentions* can be used as an organizing principle of static causal models for evaluating structure. An implementor's intentions describe how an implementor might connect primitive structures in order to achieve a desired behavior. One should note that system models for evaluating structure are not common. In fact, the only system which employs models of intentions to derive structure is PROUST [Shortcliffe76], a software diagnosis system. Non-causal models are more common, but again only in the field of software diagnosis as employed by systems such as FALOSY [Sedlmeyer83].

2.8 Expectations Models for Diagnosis

Behavioral diagnosis requires a dynamic model which describes the behavior of the artifact and a dynamic model which describes behavioral expectations. Structural diagnosis requires dynamic models of artifact structure and structural expectations. Static system models are normally used to represent system expectations. The static expectations model is submitted to an evaluation mechanism with initial conditions being the same inputs as applied to the faulty artifact. The resulting dynamic model is then compared to a dynamic model constructed from observations of the artifact

using a suitable diagnostic process.

The choice of the static model of expectations from which the dynamic model is generated affects a number of different aspects of the entire diagnostic system. This section focuses on these issues and describes the relative advantages and disadvantages of each⁸. Issues concerning the construction of dynamic artifact models are considered in the following section.

2.8.1 Diagnosis Using Non-Causal Models

Dynamic non-causal expectations models are the result of an evaluation mechanism generating possible instances of the dynamic system from a static non-causal model. The disadvantages of this approach are:

- the lack of deeper causal knowledge makes explanation of the generated dynamic non-causal model difficult;
- maintenance and modification of a large static model is difficult because the evaluation mechanism is normally tightly-coupled to the model;
- it is difficult to guarantee the correctness and consistency of the static model.

2.8.2 Diagnosis Using Causal Models

Dynamic causal expectations models are the result of submitting a static causal model to an evaluation mechanism.

The advantages of causal models are that:

- representation of the static model is easier to maintain since its organization reflects the organization of the modelled system;
- it is easier to guarantee consistency and correctness of the static model;
- the dynamic model can be explained in terms of causality – an intuitive notion.

⁸Some of these are paraphrased from [Hudlicka88].

2.8.3 Shallow Models

The advantages of utilizing shallow expectations models are that:

- they do not require a sophisticated evaluation mechanism;
- they do not require deep understanding of the system being modelled, only empirical associations between attributes of the system.

The disadvantage is that shallow models are inherently incomplete which can lead to incomplete diagnosis.

2.8.4 Deep Models

The advantage of using deep models is that a wide variety of dynamic models can be generated from the static model lending to wider diagnostic coverage.

The disadvantages are that:

- a sophisticated evaluation mechanism is required;
- evaluation may take a long time;
- in-depth knowledge of the system being modelled is required;
- model construction can be time consuming.

2.8.5 Faulty Models

The expectations model employed for diagnosis may contain *faulty* or *fault-free* descriptions of the artifact. Expectations concerning the faults and errors of an artifact are described using shallow models. This type of expectations model, sometimes called a *fault dictionary*, explicitly enumerates the possible faults and their related errors.

The advantage of a faulty expectations model, in addition to those of a shallow model, is that the diagnosis is efficient if the dictionary size is small.

The disadvantages of this approach, in addition to those of a shallow model, are:

- collection of symptom/fault pairs to construct the dictionary is time consuming;

- updates to the model are required any time the system characteristics change;
- the use of a static model is limited to the diagnosis and prediction of faults;
- in order to diagnose multiple failures, the size of a dictionary grows beyond practical limits to explicitly accommodate all possible combinations of failures.

2.8.6 Fault-Free Models

Fault-free expectations models are deep or shallow models which represent the fault-free artifact. Its advantages, in addition to those of the type of model chosen, are:

- the static model can be used for more than diagnostic purposes *e.g.* simulation;
- no change in the model is required to diagnose multiple faults.

Its disadvantages are those associated with the type of model chosen.

2.8.7 Model Determinism

Static models may be classified as either *deterministic* or *nondeterministic*. These classifications reflect our ability to create a static model capable of simulating a system's behavior.

A nondeterministic model is a static model which cannot always be transformed into a unique dynamic model when submitted to an evaluation mechanism with a complete description of initial conditions. This nondeterminism can be due to some unaccounted for behavioral variable in the model which affects the behavior of the system.

In terms of diagnosis, if a nondeterministic model is used then a failure to generate a unique dynamic model translates to a potential inability to locate faults. Both shallow and deep models are also prone to nondeterminism depending on their level of detail.

2.9 Artifact Models for Diagnosis

Artifact models are dynamic models consisting of observations of the artifact structure and/or behavior. Depending on the diagnostic process employed an inference

mechanism may be used to infer details of a poorly detailed artifact model based on the system's expectations model. However, one must ensure that fault-free structures are not inferred to be present when, in fact, they are faulty. We will examine some of these techniques in Chapter 6. This section examines a number of pragmatic issues affecting the construction of the artifact model.

2.9.1 Observability

For behavioral diagnosis, one would like to be able to observe any behavioral variable of the artifact. For structural diagnosis one would like unrestricted access to the artifact structures. Of course this is not always possible. For example, when diagnosing a VLSI circuit one cannot always determine the values of internal behavioral variables or the internal structures of the circuit. Only the values of behavioral variables at its external connectors are measurable.

There is the added problem of the act of observation affecting the behavior of the artifact. Sometimes referred to as the *probe effect*, systems sensitive to observation may behave differently when observed than when not observed. This occurs because the presence of probes in the artifact effectively alter the structure of the artifact. Thus, diagnoses undertaken with probes in place only apply to the structure of the artifact with the probes in place. Once the probes are removed, the structure of the artifact will change and new errors may be observed.

2.9.2 Reproducibility

Reproducibility is a behavioral characteristic of an artifact. An artifact with reproducible behavior is guaranteed to produce the same behavior during successive system runs. Theoretically, if the initial conditions and inputs to an artifact can be fixed, then the behavior of the artifact is reproducible. Unfortunately, neither the inputs nor the initial conditions of some artifacts can be isolated. This then, becomes the criteria for categorization: artifacts for which the inputs or initial conditions cannot be fixed are considered to exhibit non-reproducible behavior. Conversely, artifacts for which the inputs and initial conditions can be fixed exhibit reproducible behavior.

From a diagnostic point of view, reproducing erred behavior in a system is crucial for fault location when a complete model of artifact behavior cannot be constructed when the erred behavior is first observed. When this is the case diagnosis becomes an iterative process or *cyclic* and as such, requires the same fault to occur in the artifact as many times as it takes to locate the fault through successive observations. If different system behavior were to occur between successive observations, contradictory results may well arise invalidating the diagnosis. This is the reason why intermittent faults are so difficult to diagnose.

An alternative to guaranteeing reproducible behavior is to accumulate a record of an artifact's behavior leading up to the observation of an error when the error is first observed. The detail of recorded behavior will determine the observability of the system. This form of diagnosis is sometimes referred to as *post-mortem*⁹ or *retrospective* diagnosis.

2.10 Summary

The possibilities for combining different diagnostic processes with diagnostic models are numerous. The suitable choice depends on the diagnostic domain — the availability, completeness and format of the expectations models and the characteristics of the artifact. The next chapter reviews a number of software diagnosis systems which illustrate some of the combinations used in different situations.

⁹Although this is implied by the term, the system need not have incurred a *fatal error* to undergo post-mortem diagnosis.

Chapter 3

A Review of Software Diagnosis

The presence in programs of bugs can be regarded as a fundamental phenomenon — the bug free program as an abstract theoretical concept like the absolute zero of thermodynamics, which can be envisaged but never attained. [Schwartz71]

As discussed in the introduction, debugging consists of both fault location and fault correction. When locating faults in software a diagnostician may use a variety of strategies incorporating elements of both behavioral and structural diagnosis. Though a structured approach to debugging has yet to be accepted by the programming community at large, much research has been devoted towards determining how debugging might be done more efficiently.

In response to this lack of formalism researchers have studied more closely the cognitive processes utilized by expert debuggers. Researchers have developed tools to assist a diagnostician in localizing bugs more efficiently. They have proposed formal debugging procedures with clearly defined limitations. The following sections discuss the results of some of this research and how they relate to the concepts of diagnosis presented in Chapter 2. The chapter concludes with our choice of a diagnostic procedure and diagnostic models for the concurrent software domain.

3.1 Cognitive Process Models

Attempts at learning how diagnosticians locate faults in software have taken the form of formal psychological studies and informal empirical observations. The goal has been to characterize the cognitive processes of the diagnostician so that the process may be taught to novice diagnosticians or that tools might be constructed which aid the diagnostician or act altogether autonomously.

These studies concentrate on the debugging process and not so much on the cognitive models of the software the diagnostician uses. The findings of these studies indicate that software diagnosticians tend towards behavioral diagnostic techniques over structural ones. The studies also indicate that the variety of approaches taken depend on the cognitive models of the software utilized by the diagnostician.

3.1.1 Psychological Studies

Gould [Gould75] proposes a *gross descriptive model of debugging* shown in Figure 3.1. Debugging tactics are debugging procedures programmers use when localizing a fault. Although Gould did not enumerate the possibilities his study indicates structural diagnosis as the most common tactic. Gould found that “in debugging, subjects develop a set of assumptions about the correct state of the program to which the actual state is compared.” This remark implies the use of fault-free expectations models for diagnosis. He also found that programmers with access to interactive debugging facilities seldom used them and concentrated on examination of the faulty program listing¹. Finally, he found that programmers rarely used input data associated with the observation of a bug and only occasionally utilized the erroneous output data. Behavioral diagnostic techniques were not popular with Gould’s test subjects.

Weiser [Weiser82] found that programmers use a debugging technique he refers to as *program slicing*. Program slicing is the process of stripping a program of statements which do not influence a given variable at a given statement. Figure 3.2 shows an example of a code fragment and a program slice on one of its variables. His study

¹This finding must be seen in its historical context. Interactive debuggers of the day were *not* easy to use.

Figure 3.1: Gould's gross descriptive model of debugging

Code Fragment: 1) GET(A);
 2) L := A**2;
 3) Y := L*2;
 4) Z := L+2;
 5) PUT(Y,Z);

Slice on Variable *Z* at Line 5: Lines 1, 2, 4 and 5

Figure 3.2: A program slice on the variable *Z*

confirms the hypothesis that "...debugging programmers, working backwards from the variable and a statement of the bug's appearance, use that variable and statement as a slicing criterion to construct mentally the corresponding program slice." His results indicate a preference for behavioral diagnosis where slices represent deep causal models of the program artifact.

3.1.2 Empirical Studies

Schwartz [Schwartz71] points out that “in debugging, one can normally reason back, following a chain of defective entries on a time-path that is long but not spatially extended, and without ordinarily having to consider the global structure of very complex entities.” Schwartz stresses that “debugging is much more an exploration along the time than along the space dimension of the defective computation.” His observations indicate a behavioral diagnostic approach utilizing causal models of expected and actual program behavior.

LeBlanc [LeBlanc87] observes that “the most common methodology today is *cyclic*: the program is executed until an error manifests itself, the programmer postulates a set of underlying causes for the error, trace statements or additional breakpoints are inserted to gather more information about the causes of the error, and the program is re-executed.” The cyclic approach involves the use of an interactive debugger for tracing and setting breakpoints. This approach is also a form of behavioral diagnosis – the debugger acting as a flexible tool to aid the diagnostician in the construction of a deep causal model of actual program behavior.

3.2 Debugging Tools

Numerous debugging tools have been proposed to assist the diagnostician in localizing a software fault or replacing the diagnostician altogether. These tools are categorized as *interactive* tools and *autonomous* tools respectively. Interactive debugging tools are further categorized as either *passive* or *active* tools [Ducasse88]. The categorization refers to the level of involvement of the diagnostician in locating a fault. Passive debugging tools query the user for information but do not allow the user to directly control the tool when it is localizing a fault. Active debugging tools relinquish control to the user and only provide the user with guidelines for fault localization.

Not all debugging tools are diagnostic systems (*i.e.* structural or behavioral diagnostic systems) although they all contribute in some way to the debugging process. For instance, some tools provide insight into the observed behavior of the artifact. Error detectors, normally considered testing tools can be used in debugging

if they are flexible enough to be of help in locating the cause of the error.

Debugging tools may also be *loosely* classified by their flexibility with respect to the errors they help to detect or the faults they help to locate. For the purposes of discussion here, a debugging tool is either *fixed* or *flexible*. Fixed tools assist the user with debugging *specific* errors and faults. Flexible tools assist the user with debugging a *wide variety* of errors and faults.

The following sections survey existing or proposed active, passive and autonomous debugging tools with respect to their flexibility. Most of the tools are applicable to a specific software domain; either sequential or concurrent.

This is intended to be a general overview, not an exhaustive survey of the area. A detailed survey of systems applicable to concurrent software can be found in [Cheung87] and [McDowell87]. A survey of the sections on flexible passive and autonomous tools (see §3.2.3 and §3.2.4) using a different taxonomy can be found in [Ducasse88] and [Seviora87].

The terminology introduced in Chapter 2 will serve as a basis for the overview. Tools categorized as behavioral or structural diagnostic systems possess the associated advantages/disadvantages discussed in Chapter 2.

Although some of the tools discussed here are capable of fault detection and fault correction in addition to fault localization, the discussion focuses on fault localization. The sections pertaining to autonomous and passive flexible debugging tools (see §3.2.3 and §3.2.4) are the most pertinent with respect to the goals of our research. Tables summarizing the salient points of these last two sections follows the discussion (see Figures 3.3 and 3.4).

3.2.1 Fixed Autonomous Debugging Tools

These tools can be either behavioral or structural diagnostic systems. The behavioral systems analyze the output of the program artifact or make inferences from the static code artifact regarding program behavior to detect errors and locate faults. The structural systems analyze the code of the system being diagnosed to detect and locate faults. Most of the systems are primarily error detectors.

Compilers

Compilers are considered to be structural diagnostic systems. They are capable of detecting syntax errors in a program and sometimes locating their associated faults. Some compilers also provide a limited amount of semantic analysis. For example, data type checking, detection of peculiar (although legal) code constructs and data flow analysis (described in the following section).

Data Flow Analyzer

Data flow analysis is a behavioral diagnostic process used to detect specific semantic errors in a program and sometimes locate the associated fault. Data flow analyzers are incorporated as elements of compilers or exist as stand-alone programs such as LINT [Johnson86]. Data flow analysis techniques were first developed for code optimization in compilers and later extended to determine the presence of data flow anomalies. In simple terms, "Data flow analysis does not imply execution of the program being analyzed. Instead, the program is scanned in a systematic way and information about the use of variables is collected so that certain inferences can be made about the effect of these uses at other points in the program." [Fosdick76] The scanning and inferences constitute a behavioral evaluation mechanism which generates a dynamic model. Faults and errors are identified by comparison of the dynamic model to what amounts to a fault dictionary.

Some examples of faults locatable using this technique are: unused variables and functions, use of variables before they are assigned values and unreachable portions of a program. These faults are locatable in sequential and concurrent programs alike. Data flow analysis can be extended to detect a number of characteristic errors in concurrent programs such as: concurrent reference and definition of a shared variable by different tasks [Taylor80] and errors in the synchronization structure which result in the permanent blockage of a task [Taylor83]. These techniques are restricted to analyzing program interactions with static program objects—dynamic objects such as dynamically created tasks or dynamically allocated memory are ignored. This technique also requires examination of all possible program states and can become computationally intractable for large programs.

Memory Bounds Check/Exception Handling

A memory bounds checker is sometimes present in a program to monitor memory references by the program at runtime. It is capable of detecting invalid memory references such as referencing non-existent elements of an array. Programming languages or operating systems supporting their execution can also monitor for exceptional computations either defined by the user or pre-defined by the operating system such as 'divide by zero'. Both systems are primarily error detectors.

Deadlock Detector

Deadlock detectors are error detectors which monitor the execution of a program in a manner similar to the systems described above. However, deadlock detectors are capable of detecting a more subtle form of error characteristic of concurrent programs.

Helmhold and Luckham's monitor [Helmhold85] detects a class of *deadness errors* independent of program semantics in Ada programs. A deadness error occurs when part of a concurrent computation can no longer proceed due to task communication failure. Upon detection of a deadness error, the user is presented with 'snapshot' of the dead program state soon after the error is detected. From this information the user must determine where the fault lies. No history of events leading up to the detection of the error is kept. To discover what actually occurred, a programmer must re-execute the program while stepping through task interactions to construct a mental dynamic model of the program's erred execution (a feature provided by the monitor). However, re-executing an erred program is viable only for programs exhibiting reproducible behavior.

ANNA/TSL

The Stanford Program Analysis and Verification group has developed a number of tools for debugging Ada programs from their specifications [Luckham86]. For a subset of sequential Ada programs a formal behavioral specification language called ANNA has been developed. A development tool annotates Ada programs as they are developed to formally specify behavior not expressible through the Ada language

itself. Ada/ANNA programs are converted to self-checking Ada programs by an ANNA Runtime Checker [Rosenblum86]. The converted program consists of the original Ada program combined with executable checks inserted at various places to verify annotations. The runtime checks inserted by the transformer call a symbolic debugger when they are violated.

TSL is another formal behavioral language used to annotate multi-task Ada programs. The runtime monitor for TSL works similarly to that for ANNA but, to date, only an abstract implementation exists [Helmbold88]. Both the ANNA and TSL runtime monitors are error detectors.

3.2.2 Flexible Active Debugging Tools

These are behavioral diagnostic assistants which allow the user to gather the information necessary to develop dynamic artifact models, generate fault hypotheses and refine the set of suspect faults.

Replay

Replay tools enable a user to repeatedly re-execute a normally nonreproducible program in a reproducible manner. This is especially useful for locating faults as repeated execution lends itself to the construction of detailed artifact models for behavioral diagnosis. Replay tools utilize combinations of event monitoring or data state checkpointing during the initial execution of a program to record the information necessary for execution replay. So far replay tools have been restricted to concurrent software applications which do not utilize interrupts.

One replay tool for concurrent software running on a tightly coupled multiprocessor [LeBlanc87] records the partial order of all shared data access events but does not record the data associated with the events. For the sake of simplicity, LeBlanc restricts his "Instant Replay" tool to non-interactive programs. Event history based systems such as this one require that the program be replayed from the program's initial state.

Mechanisms utilizing a checkpoint technique save the execution state of the individual program tasks at periodic intervals. IGOR [Feldman88] uses a technique similar to demand paging which determines when to archive 'dirty' program memory

pages of a sequential program. BUGNET [Wittie88] provides a checkpoint replay tool that replays execution of concurrent programs on a loosely coupled multiprocessor network at close to the same speed as they were originally monitored at. This form of checkpoint mechanism requires that all IPC messages and I/O be captured in logs and timestamped.

PPD [Miller88] implements a checkpoint mechanism that is combined with a static *flow graph* of the program to be replayed. A static program flow graph is essentially a deep static causal model of a program which represents the variable dependencies between code structures (the program flow graph is investigated in more detail in §4.6.2). This enables the tool to present the execution history of a program in terms of the actual variable dependencies which existed when the program executed. The user may identify a dependency of interest and display the dependencies which preceded or succeeded it during execution. This tool is well suited to behavioral diagnosis as it provides the user with a dynamic causal model of the program and a means to explicitly examine its dependencies.

In general, checkpointing replay tools must store more information than event history based tools but allow the flexibility to replay from intermediate points in the program's execution.

Event Clustering/Event Filtering

A well-known debugging tactic used by programmers in search of a bug is embedding print statements at strategic locations in the erroneous code so that internal behavior can be monitored. A more powerful and flexible mechanism for accomplishing the same goal is event monitoring. Event monitoring systems allow the user to define event clusters of interest and filter the event stream to monitor only those events deemed interesting.

An event cluster is a chosen set of events combined into a single composite event. Clustering events abstracts detailed event information for better user comprehension of program behavior. Event filtering is a technique for reducing the large amounts of data that might otherwise be recorded during the event monitoring process.

Indicative of these techniques is the EBBA (Event Based Behavioral Abstraction) system [Bates88]. EBBA provides the user with a language for specifying event

clusters of interest in loosely coupled multiprocessor systems. Occurrence of these events is reported at runtime and a history of these events is saved for post-mortem analysis. Similar systems are outlined in [Baiardi86] and [Bruegge83].

These facilities provide a means for diagnosticians to build dynamic models of actual program behavior to compare with their models of expected program behavior.

Symbolic Debugger

A symbolic debugger is a tool which aids the diagnostician in behavioral diagnosis of a program. It provides information regarding actual program behavior in terms of the language the program was written in².

A symbolic debugger traditionally offers the user a host of functions useful for gathering behavioral information during program execution. For sequential debuggers, a comprehensive list of desired functions is given in [Seidner33]. Some of these are:

Breakpoints – to suspend program execution at specified points or for prespecified events.

Stepping – permits a program's execution to be halted every n steps.

Symbolic Variables – provide the ability to display and set values of data scalars, arrays, structures and constants.

Tracing – to display the flow of a program's execution as it executes,

Traceback – to show the path by which the current statement was reached.

In the realm of concurrent programs the functions offered by sequential debuggers are suitable for investigating the behavior of individual tasks but can offer no information regarding intertask behavior. In addition, some of the functions described above are not readily extended to a concurrent software system, either uniprocessor or multiprocessor. For example, the concept of a global breakpoint in a concurrent system is difficult to define as is the meaning of 'stepping' through system execution.

²This is accomplished by creating a symbol table of the program at compilation time and updating the symbol table runtime.

Some proposed extensions include the ability to examine/modify IPC messages, the ability to control the task scheduler mechanism and the ability to control the clock used for timeouts. Other extensions include some of the other interactive tools already discussed such as event abstraction and execution replay.

Numerous debuggers have been implemented to deal with these problems. For a thorough review of these one is referred to [Cheung87] and [McDowell87].

MAP

MAP [Tischler83] is a tool somewhat like the PPD replay tool discussed earlier in this section. The tool relies on a database containing the results of static flow analysis of a sequential program. It provides the user with the facility to identify data dependencies associated with variables of interest in the program. Unlike PPD, MAP does not monitor to determine the actual execution path taken by the program at runtime but only shows the possible paths or dependencies associated with those paths. This abstract view of possible program dependencies can be invaluable when constructing the actual and expected dynamic causal models.

3.2.3 Flexible Autonomous Debugging Tools

These are systems which generally utilize shallow expectations models – either faulty or fault-free and primarily rely on structural diagnosis.

PROUST

PROUST [Johnson84] is a debugging system intended to detect, locate and suggest corrections for bugs in PASCAL programs written by novice programmers. The PROUST approach to debugging is termed *intention based diagnosis* which is a structural diagnostic approach to debugging utilizing both deep causal and shallow non-causal models of expectations. The system attempts to recognize stereotypical *programming plans* which constitute a potentially faulty program. A programming plan is a well known faulty or fault-free coding construct.

PROUST attempts to synthesize the program being debugged from the μ -plans using an informal description of program expectations for guidance. If it encounters

an unfamiliar code construct in the artifact it attempts to find a faulty plan construct which matches it. For the plans that PROUST knows about, it can differentiate between correctly and incorrectly implemented programs.

PUDSY

PUDSY [Lukey80] is a system for debugging subprograms (procedures, subroutines, functions etc.). It combines two diagnostic approaches, one of which is similar to the plan recognition employed by PROUST. The second approach is employed for plans which are not recognized. This approach involves symbolically evaluating [Boyer75, King76] the unrecognized plans.

I/O assertions describing the recognized and unrecognizable plans are combined and compared to a I/O assertions describing the subprogram's intended behavior in effect, a deep causal model of program expectations. A logical comparison mechanism is able to locate faulty assertions and therefore, locate the code associated with them.

The symbolic evaluation mechanism is a behavioral diagnostic technique which effectively extends the fault coverage of the system beyond those bugs contained in the fault dictionary utilized during structural diagnosis.

FALOSY

FALOSY [Sedlmeyer83] is a debugging system which employs three different fault localization tactics: *fault driven*, *function driven* and *computation driven* localization. The first two tactics are primarily structural diagnostic techniques using non-causal shallow models similar to PROUST. The last technique utilizes behavioral diagnosis.

Of the three possible tactics a suitable one is chosen by an inference procedure which compares the type of output discrepancy observed with a rule base containing associations between discrepancies and diagnostic tactics.

Fault driven localization utilizes a fault model consisting of application dependent fault hypotheses associated with well-known output discrepancies. This stage of localization is a behavioral diagnostic process drawing on a non-causal shallow fault model. A fault hypothesis is tested by invoking a code recognition system similar to PROUST's which recognizes implementations of stereotypical code prototypes.

A fault is confirmed by comparing fault characteristics from a fault dictionary with the characteristics of an instantiated prototype.

Function driven localization uses the knowledge of the type of output discrepancy to determine non-causal fault-free prototypes which might have been incorrectly implemented and then proceeds to instantiate these prototypes in a fashion similar to PROUST. An incomplete instantiation is considered a fault.

The last tactic, computation driven localization, is an application of Weisser's *program slicing* technique (see §3.1.1). The program slices affecting the value of the variable observed to be in error are isolated and interpreted by the prototype recognizer. A discrepancy between the type of output and the expected type of output identifies the fault.

FALOSY, though reliant on non-causal shallow models, does offer a selection of tactics in an attempt to optimize the task of fault localization. Incorporating knowledge of program applications with knowledge of typical faults is another technique FALOSY utilizes to reduce the fault search space.

LAURA

LAURA [Adam80] is a structural diagnostic system utilizing a non-causal deep model of expectations. The model is, in fact, a correct implementation of the program being debugged which is converted into a language independent form much the same as in SNIFFER (see §3.2.4) and compared to the similarly transformed program to be debugged.

The system compares the two models applying transformations to recognize implementations of the same functions between the two programs. Any unidentified segments of the program being debugged are considered faulty. Some common differences between models are associated with well-known faults for the purposes of fault explanation. These constitute a non-causal shallow fault model of expectations.

TALUS

TALUS [Murray87] is a debugging system intended for diagnosis of small LISP programs. It is a behavioral diagnosis system which incorporates a causal deep model of expectations. The model consists of a set of correct implementations of

the program being diagnosed. The model is supplemented with sample symbolic I/O assertions. The comparison mechanism is a Boyer-Moore theorem prover [Boyer75] which locates faults by attempting to prove computational equivalence between the model programs and the potentially buggy program.

PHENARETE

PHENARETE [Wertz82] is another debugger for small LISP programs. It combines both structural and behavioral diagnosis. The structural diagnostic component employs non-causal shallow models of fault-free program constructs and typical faults. The behavioral diagnostic component uses an evaluation algorithm which assigns symbolic values to the variables in the code being analyzed and then evaluates the code on these symbolic arguments for the purposes of comparison with the expectations model.

Diagnosis Module (DM)

DM [Hudlicka88] is the diagnostic component of a distributed expert problem solving system. State transition diagrams provide causal models of expected behavior. Cooperating experts on each processor of the problem solving system communicate via blackboards and, as a result, information left behind on the blackboards serve as a dynamic model of the program artifact. DM also employs a causal shallow fault model which is used for final identification of a fault.

Behavioral diagnosis enables the system to trace an observed error back along the actual causal pathway until a predefined fault is found to fit the actual behavior. DM simulates the effects of an identified fault using the model of expected behavior to determine other symptoms it might have caused. This may account for other observed errors or indicate that other faults exist.

DM is capable of qualitative reasoning about behavioral values for special diagnostic cases in the absence of state transition diagrams. In lieu of a diagram, DM looks on the blackboards for an example of correct behavior and uses this as an expectations model to compare with the erroneous behavior.

MTA

MTA [Gupta84] is an error detection/fault location system for realtime concurrent software. It uses the CCITT Specification and Description Language (SDL) [Rockstrom82] to specify expected behavior of the tasks which comprise the software. MTA requires as input a trace of all the messages exchanged between tasks for some arbitrary amount of the software's execution. The trace serves as a dynamic artifact model. A large part of MTA is given to sorting the trace information and detecting erred behavior. Erred behavior is identified by comparing the trace accumulated for a task to possible expected behavior as specified.

Once errors have been detected MTA utilizes a causal shallow fault model to localize any faults. Errors are detected when MTA fails to deduce the next state of a task from the message trace. The fault dictionary makes explicit the behavioral localization strategy of tracing backwards along causal paths. MTA is capable of locating faults to the task level of structural abstraction.

3.2.4 Flexible Passive Debugging Tools

The following systems are primarily behavioral diagnostic systems which rely on the user to detect the initial error and provide an expectations model for the faulty program. Most of these systems require either reproducible behavior of the faulty program or a detailed record of the program's execution history as a dynamic model of the program artifact.

Knowledge-Based Program Debugging (KBPD)

KBPD [Harandi83] is a behavioral diagnostic system for locating faults encountered during program compilation. The system utilizes a non-causal shallow fault model in conjunction with a model of the actual erred program and information gathered from user queries.

The static artifact model takes the form of a flow graph derived from static analysis of the erred program. The system employs behavioral diagnosis guided by the possible causal paths derived from the flow graph to locate a fault which matches one contained in its fault dictionary.

SNIFFER

SNIFFER [Shapiro81] is a debugging system developed as part of the Programmer's Apprentice project at MIT [Rich88]. SNIFFER takes a primarily structural diagnostic approach but constrains the fault search space by utilizing execution information derived from the program being diagnosed.

Upon discovery of an unexpected result during program execution, the user invokes SNIFFER and identifies the region of code most probably containing the error and the variable in error.

The identified region of code is first converted to a language independent form called a PLAN [Rich81] much the same as in LAURA. The PLAN representation is partitioned into fragments which have stereotyped behaviors (analogous to PROUST's stereotypical plans). At this point the plans and a record of the execution history of the program are compared against a non-causal dictionary of known faulty plans.

Behavioral diagnosis is used initially to identify suspect code fragments. Structural diagnosis is used to recognize which of the fragments is flawed. SNIFFER's requirement that the user identify the erroneous code section helps to keep fault localization efficient.

Error Locating Assistant (ELA)

ELA [Korel86] is a behavioral diagnostic system which employs a fault-free *oracle model*³ of expectations. The system incorporates a symbolic debugger for user examination of variable values while stepping through program execution.

ELA is based on the ideas of Weisser's *program slicing* (see §3.1.1). The system uses a program flow graph as a deep causal model of the program artifact. Once the user identifies the variables observed to be in error, the system traces back through the possible control paths to program instructions which influence the erroneous variables as described by Weisser's program slicing technique. By placing breakpoints at these suspect instructions and executing the program until their execution, the user can determine if the variable values at the suspect lines are those

³The oracle being the diagnostician until a superior substitute is found

expected at that point in the execution. The fault location process continues until an instruction is found which assigns an incorrect value to a variable.

Since the system does not maintain a program history, program execution must be replayed from the beginning with the same input values supplied as when the erroneous behavior was originally observed. ELA operates on the assumption that only a single fault is present in the program. In the interests of limiting the possible number of user queries the system is used to diagnose only small programs.

FOCUS

FOCUS [Lyle87] is another behavioral diagnostic system utilizing the concept of Weisser's *program slices*. This system reduces the number of possible faulty statements by eliminating those statements which do not influence the value of the erroneous variable and those statements which influence variables observed to have the correct value. This technique is known as *program dicing*. The system assumes the presence of a single fault and requires the additional assumption that all incorrectly computed variables have been identified.

PDS6

PDS6 [Shapiro84] is an interactive behavioral diagnostic system for Prolog programs. It is capable of locating and correcting faults manifesting any of these symptoms: program non-termination, finite failure⁴ and termination with incorrect output. Each symptom has a fault localization algorithm associated with it. The algorithms rely on an interpreter which compiles an execution history of the procedure calls that occur during an erroneous computation to form a dynamic causal model of the program artifact.

The system is interactive in the same way that ELA is. It requires an oracle to answer questions regarding the correctness of intermediate results encountered

⁴Finite failure is associated with nondeterministic programming languages. A program P is said to finitely fail when every computation of P with input x returns a correct output but no computation returns output y . For example, a program expected to return each successive element of an input list on successive calls finitely fails when it consistently fails to return the last element of the input list.

during the execution. PDS6 is restricted to a subset of the Prolog language which excludes extra-logical predicates and predicates exerting side effects.

Rational Debugging(RD)

RD [Pereira86] is another debugging system for Prolog in the vein of PDS6. It is interactive, depending on input from an oracle. However, this system takes into account the data influences exerted on an incorrect variable to reduce the number of suspected faulty procedures. RD provides algorithms to localize wrong solutions and finite failure/missing solutions.

YODA

YODA [LeDoux85] is an Ada execution monitoring and analysis system that provides the user with a flexible retrospective/post-mortem error detection/fault location facility. The system is capable of detecting runtime errors associated with concurrency such as deadness errors (*e.g.* cyclic deadlock), misusing shared data and detecting infraction of program specifications. YODA is a behavioral diagnosis system utilizing either a non-causal shallow fault model or a model provided by the user to detect errors and locate faults. A dynamic model of program behavior is maintained in the form of an event database compiled during program execution.

The system does not use any knowledge from the user concerning observed symptoms of a bug; rather, it examines the entire event database looking for events that fit a given error description. It is up to the user to formulate queries in such a way that the search space is reduced.

3.3 A Summary of Flexible Autonomous and Passive Debugging Tools

A summary of the salient points discussed in §3.2.3 and §3.2.4 are shown in Figures 3.3 and 3.4. The tables illustrate, in brief, the capabilities and resources required of these debugging tools.

System	Error Detection (D) Fault Localization (L) Fault Correction (C)	Single (S), Multiple (M) Faults	Structure (S) Behavior (B) Diagram (D)	Expectations Model *	Artifact Model *	Domain	
						Sequential	Concurrent
PROUST [Johnson84]	D, L, C (suggested)	M	S	STANC/SF - Language Dependent Plans STAC/SFF - Language Dependent Plans	DYN/NC - Code	Small PASCAL Programs	
PUDSY [Lukey80]	D, L, C (suggested)	M	S, B	STANC/SF - Language Dependent Plans DYN/C/D - Symbolic IO Assertions	DYN/NC - Code DYN/NC - Code with Symbolic IO Assertions	Small PASCAL Programs	
FALOSY [Sedlmeyer83]	L	S	S, B	STANC/SF - Language/Application Dependent Plans STANC/SFF - Language/Application Dependent Plans	DYN/NC - Code DYN/NC - OP Discrepancies	Small LISP Programs	
LAURA [Adam80]	D, L, C (suggested)	M	S	STANC/SF - Language Independent Plan Differences DYN/NC - Model Program	DYN/NC - Code	Small LISP Programs	
TALUS [Murray87]	D, L, C (suggested)	M	S	DYN/NC - Set of Model Programs with Symbolic IO Assertions	DYN/NC - Code with Symbolic IO Assertions	Small LISP Programs	
PHENARETE [Wertz82]	D, L, C (suggested)	M	S	STANC/SFF - Language Dependent Plans STANC/SF - Language Dependent Plans	DYN/NC - Code DYN/NC - Code with Symbolic IO Assertions	Small LISP Programs	
DM [Hudlicka88]	L	M	B	DYN/C/D - Symbolic IO Assertions STAC/SF - State Transition Diagram Fragments STAC/D - State Transition Diagram	DYN/NC - Blackboard Messages		Large LISP Blackboard Expert System
MTA [Gupta84]	D, L	M	B	STAC/SF - SDL Fragments STAC/D - SDL	DYN/NC - Inter-task Message Trace		Large Real- time, Multi- task Programs

* Dynamic (DYN), Static (STA), Causal(C), Non-Causal(NC), Shallow (S), Deep(D), Faulty (F), Fault-Free(FF)

Figure 3.3: A Summary of Flexible Autonomous Debugging Tools

System	Error Detection (D) Fault Localization (L) Fault Correction (C)	Single (S)/ Multiple (M) Faults	Structure (R) Primitive (R) Diagram	Expectations Model *	Artifact Model *	Domain	
						Sequential	Concurrent
KBPD [Harand83]	L	M	B	STA/NC/S/F	DYN/NC - Oracle STAC/D - Code with Program Flow Graph	PASCAL Programs	
SNIFFER [Shapiro81]	L, C <i>in progress</i>	S	S, B	STA/NC/S/F - Language Independent Plans	STA/NC/D - Code DYN/C - Complete Execution History	Small LISP Programs	
ELA [Korel86]	L	S	B	DYN/NC - Oracle	DYN/NC - Execution Replay within a Symbolic Debugger	Small PASCAL Programs	
FOCUS [Lyfe87]	L	S	B	DYN/NC - Oracle	STAC/D - Code with Program Flow Graph DYN/NC - Execution Replay within a Symbolic Debugger	Small FORTRAN/ ADA Programs	
PDS6 [Shapiro84]	L, C	M	B	DYN/NC - Oracle	STAC/D - Code with Program Flow Graph DYN/C - Execution Replay with Monitor	Small PROLOG Programs	
RD [Perreira86]	L	M	B	DYN/NC - Oracle	DYN/C - Execution Replay with Monitor	Small PROLOG Programs	
YODA [LeDoux85]	D, L	M	B	STA/NC/S/F - Interval Logic Formulae	DYN/NC - Complete Event History		Small ADA Programs

* Static (STA); Dynamic (DYN); Causal (C); Non-Causal (NC); Deep (D); Shallow (S); Faulty (F); Fault-Free (FF)

Figure 3.4: A Summary of Flexible Passive Debugging Tools

It is interesting to note that the majority of autonomous systems primarily utilize structural diagnosis whereas the majority of passive systems utilize behavioral diagnosis. This may be due, in part, to the simplicity of specifying detailed structural models of expectations and the corresponding difficulty of specifying detailed behavioral models of expectations. In fact, the user's involvement in most of the passive systems is simply to serve as a source of behavioral expectations. Systems employing non-oracle behavioral expectations models (DM, MTA) are restricted to finding faulty structures at the most detailed level of abstraction of the model. These systems also end up relying on non-causal shallow fault models to simplify the diagnostic task but diminish fault coverage by doing so. Non-causal shallow fault models are also employed by the majority of autonomous structural diagnosis systems.

Of the remaining autonomous behavioral diagnosis systems, the majority utilize comparison of symbolic I/O assertions. All of these systems are applicable only to small LISP programs as this form of analysis is presently intractable for larger programs written in other languages.

Of the passive behavioral diagnosis systems, all but one utilize causal models of a sequential program artifact in the form of program flow graphs or execution replay with a monitor or debugger. The use of causal models here helps to reduce the fault search space.

All of the behavioral diagnostic systems developed for concurrent software rely on post-mortem analysis of execution traces. With respect to the degree of fault localization it is found that only those diagnostic systems meant for small programs are able to isolate faults to the line of code.

3.4 Applicability to Our Objectives

By considering the objectives set out in Chapter 1, the evaluation of diagnostic processes and models in Chapter 2 and the information summarized in this chapter we are able to make a preliminary decision regarding our choice of diagnostic models and a diagnostic process for the concurrent software domain.

As expectations of concurrent software are normally in terms of behavior and not

in terms of structure behavioral diagnosis is a desirable approach. For comparison with expected behavior the behavior of a program artifact is readily observable using the tools outlined in this chapter. If the diagnostic systems is interactive, the research on cognitive process models has shown behavioral diagnosis to be the most natural for the diagnostician to use.

Deep causal models are desirable as the concurrent software artifact will be modified frequently in the software implementation environment. In addition, we require the ability to diagnose a wide variety of unaccounted for faults including the possibility of multiple points of failure -- a task well suited to diagnosis using deep causal models.

The remainder of this thesis is devoted to investigating the details of applying behavioral diagnosis to deep causal models of concurrent software.

Chapter 4

Modelling Concurrent Software for Diagnosis

In the previous chapter we concluded that behavioral diagnosis utilizing deep causal models would be desirable for the purposes of diagnosing concurrent software. This chapter first explores the deficiencies of the static deep causal modelling formalism discussed in Chapter 2 as a modelling formalism for concurrent software. Following this, extensions to the formalism are proposed which accommodate these deficiencies and constitute a more general modelling formalism. The suitability of the proposed formalism is illustrated with a number of examples from the concurrent software domain. Definitions of faults and errors in terms of the new formalism are introduced with examples. We also discuss how errors might be interpreted by the observer. Finally, the proposed formalism is compared with a number of representational formalisms for concurrent software.

4.1 Extending Deep Causal Modelling

In Chapter 2 a deep causal modelling representation was introduced which consisted of a structural and behavioral description. The structural description consisted of the structures which constituted the system being modelled, the behavioral variables defined at those constituent structures and the structural interconnections. The behavioral description consisted of directed dependencies between behavioral

Figure 4.1: Deep causal model of a 2 bit adder

variables of different structures and constrained dependencies between variables defined at unique structures. A graphical depiction of this representation for a simple combinatorial logic circuit is shown in Figure 4.1¹.

This formalism is suitable for modelling combinatorial circuits but is insufficient for modelling concurrent software. Though it is able to represent the notion of concurrent causality also found in concurrent software, concurrent software is capable of exhibiting and reacting to a wider variety of behaviors than can be depicted using the formalism presented so far. To illustrate these differences, consider the sequential program artifact shown in Figure 4.2, also depicted as a deep causal model.

Software is not data driven in the way that systems represented with this modelling formalism have been up until now. Where a change in the value of C_{out1} in the two bit adder will cause FA_2 to compute new output values, a change in the value of A in the program segment will not cause the execution of S_2 . Software causes or reacts to *events* – effectively behavioral variables with no associated values. The e variable is an example of an event signalling the execution of the next instruction.

In software, none of the data variables defined at two connected structures along a causal path are necessarily interdependent. Where FA_1 and FA_2 of the two bit adder are connected by the interstructural dependency linking variables C_{out1} and C_{in2} , S_1 and S_2 of the program segment lie along the same causal path but the variables defined at these structures are independent of one another.

Software also provides for the conditional execution of code structures. For

¹To reiterate from Chapter 2, this graphical depiction shows only structures, behavioral variables and interstructural dependencies and neglects constrained intrastructural dependencies

Figure 4.2: Sequential program and its deep causal model

example, at S_3 execution may be transferred to *either* S_4 or S_5 depending on the value of D .

We propose extensions which accommodate these characteristics. However, concurrent software possesses other characteristics such as:

- the capability of dynamic creation/destruction of structures and data;
- the capability of treating structures as data;
- the capability of indirectly referencing data through pointer variables.

How to model these remaining characteristics has yet to be determined.

To accommodate the differences between deep causal models of concurrent software and the traditional deep causal models we propose three classes of behavioral variables:

event variables – boolean variables which represent the occurrence (*true*) or non-occurrence (*false*) of an event – from a causal perspective, events motivate behavior in a system;

Figure 4.3: Data variable representation

data variables – variables which can take on any value but do not motivate behavior in a system;

data-event variables – a hybrid of the two preceding classes which can take on any value and motivate any behavior but only when its value changes.

Data variables are graphically depicted as circles. This provides for the conceptual representation of ‘static’ or ‘persistent’ data rather than the ‘flowing’ concept of data present in the former representation. It also provides a representation for data abstraction as described in §4.2.

Figure 4.3 models a structure S which *references* the value of a data variable X and *defines* the value of a data variable Y as indicated by the direction of the edges d_i and d_o respectively. The edges between a data variable and a structure are in lieu of intrastructural dependencies which exist for event and data-event variables. These edges are referred to as *definition-reference dependencies*.

Intrastructural dependencies may be between any combination of behavioral variable classes but interstructural dependencies are only between variables of the same class. Intrastructural dependencies may be constrained in any suitable fashion. In Figure 4.4, intrastructural dependencies between data variables of an addition statement S_1 are expressed in algebraic terms. Inter-event dependencies can be described using a formalism capable of describing temporal relations between events such as interval logic [Schwartz83] or finite-state machines. Figure 4.4 illustrates a *while* statement structure S_2 with intrastructural dependencies described using Statecharts [Statemate87] notation; event e_1 occurs after the execution of the last statement before the loop; event e_2 causes the first statement in the body of the loop to execute; event e_3 causes the first statement following the body of the loop

Figure 4.4: Deep causal models of individual statements

to execute and event e_4 occurs after the execution of the last statement in the loop body. Note that in this formalism the occurrence of *all* input events defined at a structure is not a necessary condition for an output event to occur. Similarly, not *all* output events defined at a structure necessarily occur when the structure is executed.

With these extensions the behavioral variables of Figure 4.1 are represented by data-event variables. Data-events are used to represent data driven systems. In concurrent software behavioral variables are restricted to data and event classes. Another characteristic of software is that all dependencies are unidirectional. Software is truly causal so no causality need be imposed when constructing the dynamic causal model.

In summary, a static deep causal model of concurrent software defined with this proposed formalism consists of structures, unidirectional interstructural event dependencies representing event variables connected to the structures and unidirectional definition-reference dependencies connecting data variables to structures.

Figure 4.5: Variable assignment model

4.1.1 Deep Causal Models of Software Structures

The following sections illustrate using the extended formalism to model common concurrent software structures. We have chosen the Ada programming language [ALRM83] utilizing rendezvous IPC as the bases for concurrent software structures modelled here.

Variable Assignment

A model representing a *variable assignment* is shown in Figure 4.5. The event variable e_i represents the event which causes the assignment operation to occur. Event variable e_o represents the completion of the assignment operation.

If-Then-Else

A model of an *if-then-else* construct is shown in Figure 4.6. In this collection of structures we see the representation of conditional execution of structures. Inter-structural dependencies represent *possible* dependencies. For the *if* structure S_1 , output events e_2^1 and e_3^1 are dependent on the same data and event variables, e_1 , X and Y . However, either e_2^1 or e_3^1 will evaluate to true – not both.

For the *endif* structure S_6 either e_1^6 or e_2^6 can cause the output event e_3^6 to occur.

Accept Entry

A deep model of an *accept entry* construct is shown in Figure 4.7. The *accept entry* structure is somewhat different from the preceding structures. For the *accept E* structure S_1 , e_2^1 is a *remote event* from a calling task signifying that a caller is

Figure 4.6: If-Then-Else model

Figure 4.7: Accept entry Model

waiting on the entry. The event e_3^1 is an optional² response event directed at the same caller which signifies the call has been accepted. Note that the data variable X' is received from the remote caller task and defines data variable X for use by local structures.

For the *end* structure S_3 , event e_2^3 is sent to the caller to signify that the rendezvous has been terminated. Data variable Y is referenced and sent as Y' to the caller task at this point.

Call Entry

A deep model of a *call entry* construct is shown in Figure 4.8. The *ACCEPTOR.E* structure S_1 upon activation by e_1^1 signals a remote acceptor task with e_2^1 signifying it is waiting on its entry. Upon reception of the event e_3^1 which signifies the caller has been accepted that data variable X is transferred as X' . When the event e_4^1 is received this ends the rendezvous with the acceptor and the remotely transmitted data variable Y' is forwarded as Y .

²This event is important when timed entry calls are utilized

Figure 4.8: Call entry Model

Selective Accept

A model of a *selective accept* construct is shown in Figure 4.9. For the *select* structure S_1 , events e_2^1 and e_3^1 represent calls requested on entries $E1$ and $E2$ respectively. In Ada, a nondeterministic procedure determines which accept structure receives an event from a caller when there are callers waiting on both entries. This mechanism is assumed to be internal to the *select* structure. Based on this mechanism the *select* structure enables an *accept* structure by choosing to output either events e_4^1 and e_7^1 or e_5^1 and e_6^1 . In the former case structure S_2 is enabled; in the latter, structure S_4 is enabled. The accept structures are defined as previously discussed.

4.2 Model Abstraction

Abstracting a deep model suppresses detail in an effort to simplify. In the following sections we discuss different forms of abstraction for deep models.

4.2.1 Structural Abstraction

Structural abstraction of a model is accomplished by substituting abstract structures for collections of structures. An example of structural abstraction would be uniting the two full adders in Figure 4.1 to represent a single 2X2 adder structure. Note that when structures are abstracted the interstructural dependencies external to the abstracted structures remain unaltered.

Figure 4.9: Selective accept Model

Figure 4.10: Block abstraction models

A useful intraprocedural abstraction above the level of individual lines of code is the *basic block* where a basic block is a contiguous sequence of program lines with no jumps in except at the beginning and no jumps out except at the end [Aho86]. A further abstraction above this is grouping basic blocks by levels of *nested conditional* code structures. Examples of block abstraction are shown in Figure 4.10. The *while loop* body and *else* body of the left hand figure are basic blocks. Since the *while loop* is contained within the *then* body of an *if-then-else* construct the entire while loop may be further abstracted as shown in the right hand figure. Note that procedures and tasks are not examples of structural abstraction but are actually *object abstractions* to be discussed later.

Figure 4.11: Modelling data variable abstraction

4.2.2 Behavioral Abstraction

Behavioral abstraction consists of abstracting behavioral variables alone or abstracting behavioral variables together with interstructural dependencies.

Variable Abstraction

Abstracting behavioral variables alone only applies to data variables since they are represented as persistent elements of the model similar to model structures. Since software data accesses typically occur at different levels of data abstraction one cannot abstract the dependencies associated with the abstracted data variables. For example, Figure 4.11 illustrates the abstract data variable Y which consists of two data variables X and Z . At S_1 the Y variable is defined in entirety and at S_2 only the X component variable is defined.

Variable-Dependency Abstraction

As an example of abstracting both behavioral variables and interstructural dependencies, one could group the two pairs of 1 bit input lines A_1, A_2, B_1 and B_2 in Figure 4.1 to represent two input data variables A and B each capable of taking on a value between 0 and 3. Similarly, the three output lines S_1, S_2 and C_{out2} can be grouped into a single variable S which may take a value between 0 and 6.

Grouping interstructural dependencies dictates grouping of associated constrained intrastructural dependencies in some consistent manner. Thus, the constrained in

Figure 4.12: Procedure call model

trastructural dependency between the abstracted interstructural dependencies of Figure 4.1 described above could be $S = A + B$. For concurrent software the abstraction of intrastructural dependency constraints is, in general, not possible due to the complex nature of dependency constraints at even the lowest levels of abstraction.

4.2.3 Object Abstraction

Object abstraction is abstraction which encapsulates both data variables and structural elements. The Ada language provides object abstraction constructs such as tasks, packages and procedures. For example, examine the object abstraction provided by the *procedure call* construct shown in Figure 4.12. In our graphical representation a triangle represents an abstract object. Here event e_1 represents the initiation of a procedure call and event e_2 represents the conclusion of the call.

As another example of object abstraction Figure 4.13 models two tasks $T1$ and $T2$ synchronizing with a third task $T3$ via an entry in $T3$. Events e_1^{T1} , e_1^{T2} and e_1^{T3} are the events which occur upon task creation. Event $e_2^{T1, T3}$ represents the calling of a task entry in $T3$ from task $T1$. Event $e_3^{T1, T3}$ represents the acceptance at task $T3$ of the caller task $T1$. Event $e_4^{T1, T3}$ represents the end of the rendezvous. Similar interaction occurs between tasks $T3$ and $T2$ where $T2$ is the calling task and $T3$ the acceptor task. Another object abstraction might be grouping tasks allocated to the same processor on a multiprocessor computer into 'task clusters'.

Figure 4.13: Structural abstraction of three synchronizing tasks

4.2.4 Detail Abstraction

Another form of abstraction is *detail abstraction*. This form of abstraction purposefully disregards details of the model at the expense of model consistency. For example, when a procedure call is executed the parameters of the call are placed on a stack. The stack variables and the code structures which reference the local variables and define the stack variables are not usually considered in our models. This form of abstraction is inconsistent as it chooses to ignore details rather than encapsulate them.

4.3 Determinism

As we have shown in previous examples, modelling Ada IPC abstracts the underlying scheduling process. This introduces nondeterminism into the resulting models and subsequently rules out the possibility of simulating program behavior from the model. However, the Ada programming language is itself nondeterministic when scheduling IPC. For a deterministic model one would have to include the Ada run time system in the model.

4.4 Dynamic Models

As described in §2.6 a dynamic model differs from a static model in that behavioral variables are instantiated with values and bi-directional dependencies are instanti-

ated in one direction. To reiterate from Chapter 2, the static causal deep model is a representation of all possible behavioral interactions - the dynamic causal model represents one such interaction. In other words, a dynamic model of a program represents one possible execution path taken by the program.

Since software is executed in a causal manner, imposition of causality is not required. The presence of conditional evaluation structures in software can lead to dynamic models which do not include all structures from the static model. A dynamic model represents the notion of persistent data variables by including separate instances of data variables each time they are defined just as separate instances of structures are included each time they are executed.

Figure 4.14 shows a code artifact and deep model of two tasks $T1$ and $T2$ communicating with another task $T3$ via a common entry. Figure 4.15 presents a partial dynamic model of the executing software. The double vertical bars across the data reference edge between structures S_4 and S_3 of the deep model represents detail abstraction. The detail being ignored here is the reference and definition of the data exchanged during the rendezvous. These are operations carried out by the Ada runtime system.

4.5 Modelling Faults and Errors

At this point we are ready to propose definitions of faults and errors in terms of the modelling formalism for concurrent software. The definitions proposed here are meant to be as general as possible. Recall from Chapter 1 that a fault is a structural inconsistency between models of the artifact and expectations. An error was defined to be a behavioral inconsistency between models of the artifact and expectations. In more precise terms, a fault is defined to be an inconsistency between constraints on intrastructural dependencies or an inconsistency between interstructural dependencies in static models of the artifact and expectations. An error is defined to be an inconsistency between the values of behavioral variables in dynamic models of the artifact and expectations.

Figure 4.14: Artifact and static deep model of three communicating tasks

Figure 4.15: Dynamic model of three communicating tasks

Figure 4.16: Deep causal models of two program fragments

4.5.1 Faults

In static models of concurrent software, one finds interstructural dependencies between event variables and definition-reference dependencies between data variables and structures. An inconsistency between interstructural dependencies takes the form of an extra or missing dependency for a given structure of a static model in relation to a corresponding structure in the static model it is being compared with. For example, consider the deep causal models of two program fragments in Figure 4.16.

Considering S_1 of both models reveals that the data reference edge from data variable B is missing from S_1^2 . If the model for fragment 1 represents expectations then the absence of the data reference dependency from data variable B constitutes a fault in fragment 2 representing the artifact.

An inconsistency between constrained intrastructural dependencies takes the form of differences between the form of constraints for a given structure of a dynamic model in relation to a corresponding structure in the dynamic model it is being compared with. The differences do *not* include extra or missing behavioral variables in constraints as these are faults covered by the previous fault category.

As an example, again considering the static models of Figure 4.16, Figure 4.17 shows dynamic models for these fragments with initial conditions $A = 0, B = 0, Y = 4, Z = 2$. Considering S_2 of both models reveals no inconsistencies between intrastructural dependencies. However, the inconsistency between the defined value of

Figure 4.21: A program flow graph and a deep model of the same code section

for a sample section of sequential code in Figure 4.21. The deep causal model of the same code using our formalism is shown beside it.

Program flow graphs make explicit the control and data dependencies in a program. In our formalism, control dependencies are made explicit as event dependencies but data dependencies must be derived from the interstructural event flow and data variable definition-reference dependencies.

The flow graph formalism accommodates constraining data dependencies but constraining control dependencies is limited to predicates on data variable conditions. Abstraction is achieved by encapsulating nodes as a subgraph of an abstract node. This form of abstraction does not differentiate between abstraction of structures, data or objects. A flow graph of a sequential program corresponds to the

Figure 4.18: Dynamic models of two program fragments

Figure 4.18 shows dynamic models for these fragments with initial conditions $A = 0, B = 1, Y = 4, Z = 2$. Here we find inconsistencies at S_1 in the form of extra interstructural event dependencies e_3^2 at S_1^2 and e_2^1 at S_1^1 . These errors are manifested by the fault caused by the extra/missing data reference dependency from variable B in S_1 . The inconsistencies are not faults because they are not present in the associated static models. In Figure 4.17 we find inconsistencies in the form of the two different values defined for data variable X at S_2 . These errors are manifested by the fault caused by inconsistent intrastructural dependency constraints for variable X in S_2 .

Since it is the observance of an error which initiates a diagnosis, we now consider how errors defined in terms of dynamic models are manifested in reality. During the execution of a concurrent program certain behavior is observable; perhaps as text output to a terminal screen or as the motion of a robot arm. Differences in observed behavior from expected behavior constitute errors. In order to map an observed error to errors defined in terms of dynamic models one must identify the behavioral variables in the model responsible for the erred behavior.

We can differentiate between two, non-exclusive types of observable erred behavior: *data errors* and *control errors*. These correspond to errors associated with inconsistencies between data variable values and inconsistencies between event variable values respectively. Examples of possible data errors are a program printing an unexpected value on a terminal screen or a robot arm incorrectly positioning itself. Examples of possible control errors are repetitions output at a terminal without

a change in the output value or expecting a robot arm to react to some stimulus without even seeing it react.

These error categories are non-exclusive since, in certain instances, one kind of error is indiscernible from another. For example, one might conclude that a control error had occurred when the output to a terminal screen expected from a program never occurs. This might have been due to a non-printable character being output to the screen which causes the terminal to cease to accept characters to output – a data error.

To summarize, an error can take the form of inconsistencies between the values of either data variables or event variables in dynamic models of the artifact and expectations. For an observer of the artifact these errors can manifest themselves as data errors or control errors respectively.

4.6 Comparable Software Models

Static and dynamic models for software diagnosis were reviewed in Chapter 3. Of these the state-based models used by DM [Hudlicka88] and MTA [Gupta84] and the program dependence models of KBPD [Harandi83], ELA [Korel86] and FOCUS [Lyle87] are comparable to the modelling formalism presented here.

4.6.1 State-Based Models

State Transition Diagrams

State Transition Diagrams [Hudlicka88] group structures and data variables together as *abstracted objects* while representing event interactions separately as states. The choice of an object representation in DM is natural as the software being modelled is object oriented.

Figure 4.19 shows a graphical depiction of a state transition diagram representing a static program. Note that the graphical notation differs from the notation used for previous examples. In particular, circles do not represent data variables. Instead they represent *system states*. Data variables are not explicitly represented in state transition diagrams. They are implicitly represented as components of abstracted

Figure 4.19: An example state transition diagram

objects shown as triangles in the figure.

System states are the points in the system where events interact. States are linked to objects via *object-state links* in a many-to-one relationship. Event flows are associated with the states and data flows with the abstracted objects. Selected intraobject dependencies between data variables are qualitatively and symbolically constrained³.

The state transition diagrams limit constraint of intrastate dependencies between event variables to the non-temporal relations *XOR*, *AND* and *OR*. This serves as a restriction on state abstraction which Figure 4.20 illustrates. The *AND* across the output events of state S_1 in the left hand figure signifies that e_1 and e_2 are output simultaneously. Following the occurrence of e_1 , event e_3 occurs as output from state S_2 . Abstracting states S_1 and S_2 to form state S_1^* (shown in the right hand figure) is impossible since there is no way to state that e_2 occurs before e_3 using the non-temporal relations provided. The state-transition diagram is restricted to abstraction amongst states or abstraction amongst objects.

The dynamic program is modelled by instantiation of state-object pairs. State object links serve to identify the procedural element of an object involved in a state transition.

³Dependencies are symbolically constrained using path expressions

Figure 4.20: A state transition diagram that cannot be abstracted

System Description Language

Gupta [Gupta84] utilizes the CCTT Specification and Description Language (SDL) [Rockstrom82] to model behavior of individual system tasks. SDL is comprised of a small number of constructs which relate to high level constructs present in concurrent software such as sending/receiving signals between tasks, conditional direction of control flow within a task and processing data within a task. SDL provides no representation of data or data flow. In our formalism signals are represented by intertask events and control flow is represented by intratask event flow.

SDL provides an informal abstraction mechanism which consists of grouping SDL constructs connected by control flows local to the task into 'states'. States cannot be grouped into other states. No abstraction beyond this limited structural mechanism is provided. Event dependencies at SDL structures are informally defined in english. As SDL is meant to be an implementation-independent specification language its shortcomings as a deep modelling formalism are understandable.

4.6.2 Program Dependence Models

Flow Graphs

Another modelling formalism similar to ours is the program flow graph. A program flow graph shows control and data dependencies in a program and their relation to program structures [Aho86]. Primarily used as a formalism for code optimization and static analysis, flow graphs have been adopted by Korel [Korel86], Harandi [Harandi83] and Lyle [Lyle87] for debugging sequential software.

A program flow graph consists of a directed acyclic data flow graph superimposed upon a directed control flow graph. An example of a program flow graph is shown

Figure 4.21: A program flow graph and a deep model of the same code section

for a sample section of sequential code in Figure 4.21. The deep causal model of the same code using our formalism is shown beside it.

Program flow graphs make explicit the control and data dependencies in a program. In our formalism, control dependencies are made explicit as event dependencies but data dependencies must be derived from the interstructural event flow and data variable definition-reference dependencies.

The flow graph formalism accommodates constraining data dependencies but constraining control dependencies is limited to predicates on data variable conditions. Abstraction is achieved by encapsulating nodes as a subgraph of an abstract node. This form of abstraction does not differentiate between abstraction of structures, data or objects. A flow graph of a sequential program corresponds to the

basic block level deep model of the same program using our formalism.

Flow graph formalisms for concurrent programs are still being developed. Taylor [Taylor80, Taylor83] employs different formalisms for modelling control flow between tasks and data flow affecting shared variables. Miller and Choi [Miller88] describe *static and dynamic program dependence graphs* for multi-task programs which correspond to our static and dynamic models respectively. Their model is used by an execution replay mechanism described in §3.2.2. Miller and Choi's model is similar in most respects to ours including the inability to model dynamic structures and data variable pointers. However, as with sequential flow graphs, constraining event dependencies beyond simple predicates on data conditions is not supported. Also, this formalism does not represent data flow between tasks.

4.7 Summary

We have outlined a formalism for static and dynamic causal modelling of concurrent software. The formalism extends on previous diagnostic modelling formalisms by representing structures and behavior unique to concurrent software; in particular:

- structures which output events conditional upon data values or output events conditional upon the temporal relationship between other events received at that structure;
- consistent representation of the software structures at all levels of abstraction — abstraction which conveniently parallels that found in the programming language itself;
- data is represented statically — not flowing;
- event flow is decoupled from data flow.

We have also proposed comprehensive definitions of faults and errors in terms of the formalism. The next chapter investigates possible sources of the models in the concurrent software implementation environment.

Chapter 5

Concurrent Software Modelling Sources

Our goal is to implement a diagnostic system to assist in the development of Unison applications. In Chapter 3 we refined our goals by deciding to use a behavioral diagnostic process utilizing static causal deep models and dynamic causal models. A suitable modelling formalism was presented in Chapter 4 but the *sources* for the artifact model and expectations models remain to be isolated. This matter is addressed in the following chapter.

5.1 The Environment

We first examine the environment within which the diagnostic system will operate. Figure 5.1 shows the current development environment hardware configuration. The host computer is a Sun Microsystems workstation. The target is a tightly-coupled multicomputer consisting of up to 16 Motorola 68020-based DY-4 DVME-134 single board computers [DVME-134]. The single board computers communicate with one another along a VME bus. The ethernet link provides a channel for downloading from the host to the target. Programs executing on the target communicate with the host along a serial communications link.

There are numerous software tools available in this environment. The *Unison* multitask, multiprocessor operating system [Unison87a] consists of a number of C

Figure 5.1: Current development environment hardware configuration

linkable libraries. Unison applications are written in C and call Unison runtime library procedures for creating/destroying tasks, allocating/deallocating memory, binding tasks to interrupt levels, calling tasks, selectively accepting calls from tasks, replying to caller tasks and signalling tasks (this is only possible from interrupt tasks).

Libraries exist for execution of an application on the target system or on the host system. The host system libraries facilitate simulation of an application on the host before downloading to the target. The simulator, called *CopyCat* [Unison87b], also provides ‘hooks’ to Sun Microsystems’ *dbx* or *dbxtool* symbolic debuggers. In addition, *CopyCat* provides the user with the ability to trace messages between tasks, monitor task state changes and simulate interrupts. The *Remedy* symbolic debugger [Unison87b] is a multiprocessor, multitasking debugger which is used to debug application programs running on the target. *Remedy* provides all the facilities of a symbolic debugger such as *dbxtool* plus the ability to monitor the state of each task and messages between tasks on the target.

A graphical software design tool, *ARTTisan* allows a designer to graphically describe the structure of a concurrent system and describe the behavioral details as C code in a manner consistent with the structure. *ARTTisan* automatically generates the *skeleton* code describing the system structure and positions the behavioral *flesh* code within the skeleton code appropriately.

5.2 Static Causal Artifact Models

As mentioned in §4.6.2, the program flow graph comes very close to our desired static model. In fact, Miller and Choi's [Miller88] *program dependence graph* would suffice if extended to include intertask data dependencies and constraints on intrastructural event variable dependencies. Software to generate these models from the code is not part of the current environment and would have to be built.

5.3 Dynamic Causal Artifact Models

Accumulating the data to construct a dynamic causal model of the artifact requires observation of the artifact. This can be done in a cyclic manner requiring reproducible behavior from the artifact or in a post-mortem manner working from a record of artifact behavior.

5.3.1 Artifact Reproducibility

From the point of view of reproducibility, Unison applications without interrupt tasks are capable of exhibiting reproducible behavior when executing on the uniprocessor host because Unison uses a nonpreemptive scheduler. In a multiprocessor environment applications exhibit nonreproducible behavior for reasons discussed in §1.4.1. To enforce reproducibility in this environment tools such as the Instant Replay tool [LeBlanc87] (see §3.2.2) can be used but a replay tool has yet to be developed which can faithfully reproduce interrupts.

Combining the reproducible artifact with an appropriate symbolic debugger allows for detailed and reliable observations of the dynamic artifact. This is common practice for constructing dynamic artifact models of sequential software.

Another approach that might be taken is compiling a record of application behavior by monitoring its execution. In this case, the detail of the recorded information is normally insufficient for execution replay necessitating post-mortem dynamic artifact model construction. Monitoring might be achieved through tracing with a symbolic debugger or with altogether separate monitoring hardware.

5.3.2 Artifact Observability

From the point of view of observability, a symbolic debugger provides a detailed view of the dynamic software artifact. In the host uniprocessor environment, integrating dbxtool with the CopyCat simulator affects only the performance of a Unison application. However, the dynamic software artifact in a multiprocessor environment is particularly sensitive to observation. The behavior of the application observed using Melody can be quite different from its behavior observed without the debugger since the debugger affects the timing of intertask synchronization and communications. Utilizing a hardware monitor diminishes the size of the observer's window on the artifact but affects its behavior little.

5.3.3 Hardware Monitoring

Utilizing a hardware monitor which is always present in the executing application insures the monitored behavior is no different from the unmonitored behavior. This technique has been employed for a performance analysis tool by Wybranietz *et al.* [Wybranietz88]. Their philosophy is "to view testing and performance measurement not as an occasional activity, but as an ongoing process that is inseparable from the system."

The monitor system they propose is shown in Figure 5.2. The monitored system is a loosely-coupled distributed system connected by a token ring network. Their monitor system consists of *local monitors*, implemented as hardware and software, at each node in the network. The local monitors autonomously gather, reduce and summarize information from each node and transfer summaries to a central monitor station along an independent monitor network. The central station requests more information from a local monitor when required. By offloading the majority of the information management tasks to local monitors the central monitor is available to interpret the information in the context of the executing application software.

The final aspect of the system is embedding *events* deemed to be of significance to the monitor into the application software. A limited number of events such as 'task created' or 'task destroyed' are permanent parts of the application's underlying runtime system. When an event occurs, it is directed to the local event monitor

Figure 5.2: An event monitor for a distributed system

of the node on which the event occurs. With the aid of an application's symbol table, the monitor system is able to interpret and present events in an application oriented manner. Initial experiments with their system have shown an overhead in the measured system of a value lower than 0.1%.

To summarize¹, the monitoring system:

- provides realtime monitoring without changing system behavior and without significantly slowing down the performance of the monitored system;
- provides high transparency at the expense of additional hardware.

The philosophy of this system is appropriate for our needs, however, its design will differ to suit the tightly-coupled multiprocessor environment.

5.3.4 Other Issues

Tracing and monitoring mechanisms can require large amounts of memory to store the information necessary to reconstruct program behavior (particularly checkpoint based replay mechanisms). Taking into account the nonterminating nature of concurrent software applications and bounded storage space, it is realistic to assume only the last t time units of program execution will be recorded — the time since

¹From [Wybranietz88].

the program's initiation being greater than t . This implies that any dynamic causal model constructed from this information will be incomplete.

Some modelling sources such as the hardware monitor trace are inherently lacking in detail. Details regarding the occurrence of unmonitored events need to be uncovered before a suitable dynamic causal model can be constructed. Algorithms utilizing the static causal artifact model and the diagnostician can be used as sources of missing details.

5.3.5 Alternatives

Considering the preceding factors leads us to choose four possible sources for dynamic causal artifact models.

1. A model may be constructed in a cyclic fashion for an application executing on the host provided it does not utilize interrupts. This would be accomplished with dbxtool or dbx combined with the CopyCat simulator.
2. A model for an application executing on the target may be constructed in a cyclic fashion provided the application does not utilize interrupts. This would be accomplished with a replay tool combined with the Melody debugger.
3. A model for an application executing on the target may be constructed from post-mortem analysis of an execution record compiled by the Melody debugger trace facilities.
4. A model for an application executing on the target may be constructed from post-mortem analysis of an execution record compiled by a hardware monitor.

We now discuss the relative merits of each alternative.

Alternative #1

Cyclic execution of the artifact permits observation of data variable and event variable values at different structures throughout the artifact. Here the static causal model is the compiled code of the program artifact and the evaluation mechanism is the machine code interpreter of the host and the CopyCat/dbxtool debugger; but this only allows the construction of a partial dynamic causal model.

Interstructural variable dependencies can be determined from referencing the program dependence graph as can intrastructural dependency constraints between

event variables and constraints between data variables at the lowest level of structural abstraction. As constraints between data variables or constraints between event variables and data variables cannot be determined at levels of abstraction above the lowest level, these details must be left absent from the model. Thus, the program dependence graph is another static causal model of the artifact; the evaluation mechanism here being a graph traversal algorithm.

The resultant model derived from this technique has the potential of being very detailed but as artifact execution occurs on the host, diagnoses will only be applicable to errors encountered on the host.

Alternative #2

As in alternative #1, cyclic execution of the artifact, this time on the target, permits observation of data variable and event variable values. The static causal model here is the compiled code of the program artifact combined with any information compiled by the replay mechanism when it originally records program behavior. The evaluation mechanisms are the machine code interpreter of the target, the replay mechanism and the Melody debugger.

Again, the program dependence graph of the artifact is used with a graph traversal algorithm as a supplementary static causal model and evaluation mechanism respectively. To be of use for diagnosis the replay mechanism must have recorded the observed error. Otherwise the erred behavior must be reproduced and recorded without the aid of execution replay.

The resultant model has the potential of being very detailed and reflecting artifact execution on the target. However, the overhead incurred by a replay mechanism can substantially alter artifact behavior rendering diagnoses only applicable to errors encountered while recording artifact execution.

Alternative #3

This alternative allows use of interrupts in the application. The static causal model is the compiled code of the program artifact and the evaluation mechanisms are the machine code interpreter of the target and the Melody debugger. The resultant dynamic model is as detailed as the execution record which may consist of data on

2 of /de 2

Micred

event variable values at chosen structures at different levels of abstraction in the artifact.

From this, the dynamic model can be supplemented with dependency details as in alternatives #1 and #2 only in this case the detail of the execution record is fixed and cannot reliably be supplemented with details in a cyclic manner as in the previous two approaches. Thus, any ambiguities arising during the construction of the dynamic model will have to be resolved by the diagnostician. As in alternative #2, the execution record is of no use for diagnosis if an error's occurrence has not been recorded. Because the behavior of the artifact is altered by the presence of the debugger, diagnoses are applicable only for errors observed while tracing artifact execution.

Alternative #4

This alternative also allows the use of interrupts in the application. The static causal model is the compiled code of the program artifact and the evaluation mechanism is only the machine code interpreter of the target. The hardware monitor is not considered an evaluation mechanism since it is a nonintrusive observer—artifact behavior is no different with the monitor present than it is with the monitor absent.

The resultant dynamic model as derived from the execution record will consist only of event variable values at different levels of artifact abstraction. Event variable dependencies can be added to the model using the program dependence graph, however, as it is likely that the level of detail of the execution record will be fixed at a level lower than that of alternative #3, the resulting dynamic model will be ambiguous.

With hardware monitoring, the likelihood of recording an observed error is good as the monitor is part of the artifact. Diagnoses based on the execution record compiled by a hardware monitor are applicable to errors observed during normal execution of the artifact.

5.4 Expectations Models

The graphical design specification produced through the ARTTisan design tool is used as the input to a semi-automatic design to code transformation. As this is a correctness preserving transformation, the design specification cannot be used as an expectations model.

We are left with the diagnostician as the source of expectations models. The diagnostic system now takes on an *interactive* requirement. The diagnostician should be able to answer questions revealing aspects of the dynamic causal expectations model including the presence or absence of dependencies between variables and the values of event and data variables. Such detailed questions could easily discourage the diagnostician from utilizing a diagnostic tool so careful consideration should be given to issues of query complexity and query comprehensibility.

In the case that a diagnostician may reply with contradictory assertions concerning expectations, a truth maintenance algorithm should be in place to detect the inconsistency.

5.5 Summary

In this chapter we have discussed possible sources for dynamic causal models of concurrent software in a particular development environment. As shown in Figure 5.3, the four alternative sources for dynamic artifact models trade off detail for observational intrusiveness. Certainly diagnosing faults can be made more autonomous with a more detailed artifact model. On the other hand, artifact models constructed by nonintrusive means are accurate reflections of an artifact's true behavior and, as such, diagnoses using these models are applicable to the artifact executing undisturbed in its environment.

The modelling sources might be accessed in the following manner during the course of software implementation: initial debugging using alternative numbers 1 and 2 and final debugging using alternative numbers 3 and 4. This progression of models corresponds to the phases of implementation which begin with simulation of the application on the host before downloading to the target for execution.

As for expectations models, we have little choice but to resort to the diagnosti-

Model Source	Model Detail	Observational Intrusiveness
Alternative #1	HIGH	HIGH
Alternative #2	HIGH	HIGH
Alternative #3	MEDIUM	MEDIUM
Alternative #4	LOW	LOW

Figure 5.5. A comparison of dynamic artifact model sources

cian. In the following chapter we determine which aspects of a number of behavioral diagnostic approaches are most applicable to the model characteristics outlined here.

Chapter 6

Diagnostic Processes for Causal Models

Creating a bug-free program is easy; simply begin with the null program and debug it until it does what you want.

This chapter investigates possible behavioral diagnostic approaches for the causal deep modelling formalism described in Chapter 4. We first review two behavioral diagnostic approaches utilizing causal deep models: violated expectations and PDS6; the former originally meant for hardware diagnosis, the latter for sequential Prolog programs. We conclude by considering aspects of these approaches most suitable to our goals.

6.1 Violated Expectations

Violated expectations is a label applied to a class of behavioral diagnostic processes, proposed or implemented as research systems since 1976. The technique, oriented towards the domain of hardware diagnosis, requires a detailed static causal model of expectations and observations of behavioral variable values from the artifact.

6.1.1 Expectations Models

The model of expectations for violated expectations is a static deep causal model of the sort described in Chapter 2 with all behavioral variables being data-event variables. A comparison of this modelling formalism with the proposed formalism for concurrent software is contained in Chapter 4. In the domain of hardware diagnosis the model can also serve as a detailed design description or a simulation model.

A characteristic of many violated expectations systems is hierarchical abstraction of the expectations model. Diagnosing a hierarchical model proceeds by first isolating potentially faulty structures at a high level of abstraction. At the next lower level of abstraction, new fault candidates are isolated from the structures which comprise the candidates at the higher level. Imposing a hierarchy upon a system reduces the number of structures considered at a given level of abstraction. The computational advantage gained by decreasing structural complexity is offset by the corresponding increase in behavioral complexity. The advantage is lost when the effort required to evaluate behavior exceeds the effort required to consider the elements at a given level of abstraction.

At each level of abstraction, intrastructural dependencies are both explicitly constrained with equations and implicitly constrained by the intrastructural dependency constraints at the next lower level of abstraction.

6.1.2 Artifact Models

The artifact model for violated expectations consists of non-causally organized observations of data variable values at artifact structures. In violated expectations, the nature of the artifact usually limits its observability. For example, the observability of a chip is limited to the variable values at its pins.

6.1.3 The Diagnostic Process

Figure 6.1 shows the series of operations involved in a general violated expectations diagnostic process as independently put forward by Reiter [Reiter87] and de Kleer and Williams [deKleer87]. Unexpected outputs and the inputs applied to the artifact

serve as the initial conditions to an evaluation mechanism for the static causal model of expectations. The diagnostic process begins by attempting to generate a dynamic causal model. A behavioral inconsistency encountered during the evaluation process results in the formation of *conflict sets* – sets of structures involved in the behavioral inconsistency. Structures in conflict sets are not necessarily all faulty but might have been involved in propagating the error. The next stage of the procedure determines which structures must have been faulty in order for the error to occur. These sets of structures are called *candidate sets*. If only one candidate set is generated, then the faulty structures which explain the error are those structures which comprise that set. More than one candidate set entails generating tests (changing inputs to the artifact) or recommending measurements (inputs to artifact remain fixed) to eliminate candidate sets. Failure to generate a test or recommend a measurement ends the diagnosis inconclusively.

6.1.4 Evaluation Mechanisms

Much of the power of violated expectations systems lies in the evaluation mechanism employed. The evaluation mechanism should be able to evaluate the variables for given initial conditions and check for any inconsistencies in the variable values generated. When an inconsistency is encountered the conflict set consists of those structures with interstructural dependencies which influence or depend on the inconsistent variable value. Thus, the set consists of those structures on the causal path along which the inconsistency is encountered.

Resolution Theorem Provers

Reiter [Reiter87] has developed an algorithm that will compute all diagnoses (with respect to the expectations model) for a given unexpected behavior provided that a sound and complete theorem prover is used as an evaluation mechanism. For example, a complete diagnosis can be achieved using a horn clause representation of expectations with an SLD-resolution [Hogger84] inference procedure¹. Genesereth [Genesereth84] incorporates a resolution residue-based theorem prover [Finger85]

¹This corresponds to the database representation and inference mechanism of Prolog.

Figure 6.1: The violated expectations process

in his violated expectations diagnostic system. This technique uses a direct proof procedure to simultaneously evaluate behavior and generate candidates.

Constraint Propagation

Another common type of evaluation mechanism utilized in violated expectations systems is constraint propagation (see §2.4.2). First used by de Kleer [deKleer76] as a conflict recognition technique, variations of the technique have been utilized by Davis [Davis84] and again by de Kleer [deKleer87]. Most constraint propagation techniques are incomplete evaluation mechanisms. The consequence of an incomplete evaluation mechanism is detecting fewer candidates which entails the elimination of fewer candidates. This does not lead to candidates being mistakenly eliminated though. In comparison to resolution-based evaluation mechanisms, constraint propagation offers computational efficiency at the cost of diagnostic completeness.

To assist the constraint propagater in locating the causal paths along which an inconsistency may lie, de Kleer's propagator in LOCAL [deKleer76] utilizes information regarding *corroborations*. In order to take advantage of corroborations, observed variable values must be specified as either correct or incorrect before the propagator begins its evaluation. Thus, any consistencies encountered along a causal path associated with correct variable values is a corroboration which serves to exonerate the structures on the path from consideration as potential conflicting structures. This is similar to the *program dicing* technique proposed by Lyle [Lyle87] to complement Weisser's *program slicing* technique (see §3.2.4).

6.1.5 Testing and Measurements

This stage of the diagnostic process is meant to propose tests or measurements which can discriminate between faulty and fault-free candidates. This stage assumes the existence of nonintermittent faults. In practice, this aspect of a violated expectations system has been developed the least. Testing is useful when the observability of the artifact is restricted and it is easy to change its inputs. Measurement, on the other hand, is a much simpler operation if observability is not an issue.

Figure 6.2: Half-split measurement

Measurement

The most straightforward technique is the *half-split* technique. Assuming a single fault, a measurement is proposed equidistant on a causal path from two previous measurements. For example, in Figure 6.2 the next measurement would be taken at x_2 based on previous measurements having been taken at x_i and x_o . This technique has the potential of eliminating up to half the candidates generated depending upon the arrangement of dependencies between candidate structures. de Kleer employed this technique in LOCAL [deKleer76] but later went on to develop more rigorous theory of measurement in [deKleer87]. Using *a priori* probabilities of individual component failures, his system uses a method to determine a measurement which will on average lead to the discovery of the actual candidate in a minimum number of subsequent measurements.

Testing

Genesereth [Genesereth84] developed a technique for generating discriminatory tests using the same resolution residue theorem prover utilized for candidate generation in his DART system. Shirley [Shirley83] developed a test generation system meant to interface with Davis' IIT system [Davis84]. This system uses dependency information provided by the constraint propagation mechanism to generate suitable tests. Both systems produce tests which are likely to give useful information though neither pays any regard to optimizations such as minimizing the number of tests required or minimizing the cost of applying a test.

6.1.6 Fault Coverage

What is the nature of faults uncovered using this technique? The structures output by the diagnostic process have intrastructural dependency constraints which have been inferred to be inconsistent with those of the same structures in the artifact. This technique is applicable to models which explicitly state intrastructural dependency constraints rather than expressing them only in terms of constituent structures at a lower level of abstraction.

An acknowledged failing of this technique is its inability to locate faults due to extra/missing interstructural dependencies. Complete fault coverage can be guaranteed only when the structural description of the expectations model corresponds with the artifact.

This drawback is exemplified by attempting to diagnose an artifact with a short circuit. As shown in Figure 6.3, the short between two wires S_1 and S_2 exists as an unexpected structure S_3 . Connecting S_3 to S_1 and S_2 are two unexpected interstructural dependencies d_1 and d_2 . The short causes the variables of S_1 and S_2 that were once independent to become dependent. An exhaustive evaluation of the expected behavior is shown in Figure 6.4. Given the observation $I_1 = 0$, $I_2 = 1$, $O_1 = 1$ and $O_2 = 1$, comparison with Figure 6.4 indicates a conflict at O_1 . This inconsistency implies the only possible cause of the fault to be S_1 . Another possible observation might be $I_1 = 1$, $I_2 = 0$, $O_1 = 1$ and $O_2 = 1$, which would lead to the conclusion that S_2 was at fault. Neither of these conclusions is correct, however, as the real fault is due to the existence of S_3 . Of course, a short circuit is not the only way a dependency can be neglected. A model which fails to include any structure present in the artifact will introduce this problem. Similarly, the failure to account for a dependency between accounted for structures will also introduce this problem.

Diagnosing a break is a scenario identical to that of diagnosing a short only that the roles of the artifact and domain model are reversed – expected dependencies are now missing from the artifact. Davis [Davis84] has proposed an alteration to the violated expectations diagnostic process in an attempt to alleviate this problem. He proposes introducing extra dependencies (in the case of a suspected short circuit) into ‘likely’ areas of the expectations model after encountering an incorrect diagnosis. This constitutes introducing a non-causal shallow fault model of expectations.

Figure 6.3: Short circuit artifact and its model

i_1	i_2	o_1	o_2
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

Figure 6.4: Exhaustive evaluation of expected behavior

6.1.7 Assumptions

Different varieties of violated expectations incorporate different assumptions in their methods. Knowing what these assumptions are is important for, as Genesereth points out, during a diagnosis “the inappropriateness of an assumption usually manifests itself as a contradiction” [Genesereth84].

Single Fault Assumption

This assumption constrains candidate sets to a single structural element. Early violated expectations systems such as LOCAL [deKleer76], IIT [Davis84] and DART [Genesereth84] use the single fault assumption to reduce the potential number of faults considered during a diagnostic session. The assumption has since been abandoned by de Kleer [deKleer87] and Reiter [Reiter87] but only at some computational expense.

Nonintermittency Assumption

The fault is assumed to be nonintermittent thereby excluding artifacts with non-reproducible behavior. This is a common assumption for violated expectations diagnostic systems, however, a technique has been demonstrated by Hamscher and Davis [Hamscher84] that successfully diagnoses circuits with state disregarding this assumption.

6.2 PDS6

As outlined in Chapter 3, PDS6 [Shapiro84] is a diagnostic system for sequential Prolog programs utilizing an *oracle* model of expectations, the faulty program as a static causal artifact model and a program interpreter to build a *computation tree* dynamic causal model. Three algorithms are used to diagnose the faults associated with three kinds of errors:

- termination with incorrect output;
- termination with missing output (finite failure);
- nontermination.

For each error “a property of procedures is defined for which an error in the program implies that it contains a procedure having this property. This procedure needs to be modified to eliminate the error in the program” [Shapiro84]. We first discuss the models used by PDS6 in terms of the causal modelling formalism outlined in Chapter 4. This is followed by a discussion of each fault/error pair.

6.2.1 Artifact Models

There are two varieties of artifact models utilized by PDS6. The first one utilizes the faulty Prolog program as the static causal artifact model and a computation tree as the dynamic causal artifact model. A computation tree is the Prolog program’s *proof tree* [Sterling86]. However, the computation tree model is applicable to languages “in which the basic computation mechanism is a procedure (or function) call, but is insensitive to the inner workings of procedures” [Shapiro84]. A Prolog program and an example proof tree are shown in Figure 6.5. The nodes of a computation tree are triples $\langle p, x, y \rangle$ where the p ’s are procedure names and the x ’s and y ’s are input and output vectors respectively².

A dynamic model of the artifact is created from the static model using a Prolog interpreter. In comparison to the formalism proposed in Chapter 4 the triples represent structures uniquely identified by procedure names and the input and output vectors are data variable values at those structures. Grouping the data variables as input and output variables implies the directionality of intrastructural dependencies. Interstructural data variable dependencies are not represented.

A computation tree is an ordered tree such that a post-order traversal of the tree reveals the interstructural event variable dependencies. The computation tree models procedural abstractions, each successive lower level of the tree representing lower levels of abstraction. Computation structures other than procedure calls are not represented using this formalism. Intrastructural data variable dependencies of a procedure call are not explicitly constrained, they can only be inferred from the procedure calls of its constituent procedure calls at the next lower level of abstraction. Intrastructural event variable dependencies are not explicitly constrained

²The proof tree and computation tree differ in their nodal notations. The nodes of a proof tree are instantiated Prolog terms; the nodes of a computation tree are triples


```

father(zeus,athena).
father(zeus,dionysus).
male(dionysus).
female(athena).
daughter(X, Y) ← father(Y,X), female(X).
son(X,Y) ← father(Y,X), male(X).

```

Figure 6.5: A Prolog program and a possible proof tree

either but as program execution is sequential they are identical for all nodes of a tree.

The static model from which the dynamic model is generated can be nondeterministic when modelling Prolog program artifacts *i.e.* for a given triple there may be more than one computation tree rooted at that triple.

Another form of dynamic artifact model utilized by PDS6 is the procedure call stack built by the interpreter during program execution. This model represents the order of procedure invocation and, therefore, the interstructural event variable dependencies in addition to the information conveyed by a computation tree.

6.2.2 Expectations Models

There are two varieties of expectations model utilized by PDS6. The first is a dynamic non-causal model referred to as an *interpretation* $\mathbf{M}$, where $\mathbf{M}$ is a set of triples representing all possible expected instantiations of input and output vectors for the procedure calls of a program. The set is not ordered in any way and therefore contains no information concerning interstructural dependencies, nor does it contain

any information concerning intrastructural dependency constraints. There is no static model for $\mathbf{M}$ – the source of $\mathbf{M}$ is an *oracle*, either *ground* or *existential*. A ground oracle can answer queries of the form, “Is $\langle \mathbf{p}, x, y \rangle$ in $\mathbf{M}$?”, where x and y are ground. An existential oracle can answer queries of the form, “Is $\langle \mathbf{p}, x, y \rangle$ in $\mathbf{M}$ and if so, what are all possible expected values of x and y ?”, where x and y are uninstantiated. Note that $\mathbf{M}$ does not represent a single dynamic model of expectations but *all* possible dynamic models for a program.

Another expectations model represents a *well founded ordering* relation $\succ$ on the set $\mathbf{M}^3$. The well founded ordering on $\mathbf{M}$ represents the expected interstructural event variable dependencies for all possible dynamic models of a program. The source of the $\succ$ relation on $\mathbf{M}$ is an $\succ$ *oracle* which can answer queries of the form, “Is $\langle \mathbf{p}, x \rangle \succ \langle \mathbf{q}, u \rangle$?” where x and y are ground.

6.2.3 Termination with Incorrect Output

An *incorrect output error* is the observation of an output data variable value at a structure that is inconsistent with the value expected at that structure during the execution of a program which terminates. Shapiro attributes such an error to the existence of an *incorrect procedure* in the artifact. An incorrect procedure is one whose constituent procedures in the dynamic artifact model return expected outputs but does not return an expected output itself.

The diagnostic process, a single stepping algorithm, is interleaved with the generation of a dynamic causal model of the artifact from the faulty program (the static causal artifact model) with the input and erroneous output values as initial conditions. The evaluation mechanism is a Prolog interpreter modified to temporarily halt execution each time a procedure call returns. When a procedure returns from being called its corresponding fully instantiated triple is checked for membership in the dynamic expectations model $\mathbf{M}$. If it is a member, the diagnosis proceeds; if it is not, this is the incorrect procedure. An example of this procedure is included in Appendix A.

³A *well founded ordering* $\succ$ on $\mathbf{S}$ is a strict partial ordering on $\mathbf{S}$ that has no infinite descending sequences. That is, $\succ$ is a binary relation over $\mathbf{S}$ which is transitive, asymmetric and irreflexive, such that for no infinite sequence $x_1, x_2, \dots$ of elements of $\mathbf{S}$ do we have that $x_1 \succ x_2 \dots$ [Shapiro84]

The algorithm must query a ground oracle to check membership in $\mathbf{M}$. For this algorithm, an upper bound on the number of queries is equal to the number of triples in the computation tree. Shapiro proposes an alternative algorithm utilizing a half-split strategy (see §6.1.5) to set a lower bound on the number of queries to be $\log_2$ of the number of triples in the computation tree. Pereira [Pereira86] attempts to improve on this strategy by introducing interstructural data variable dependencies into the dynamic model of the artifact. Thus, only structures with data variables which influence the incorrectly valued data variable need be checked. Pereira's improvement imposes *necessary* conditions for a query as opposed to Shapiro's *sufficient* conditions for a query. Shapiro also suggests storing previous oracle replies to reduce the number of queries.

Because the intrastructural data variable dependencies are not constrained, the procedure of a triple which is not in $\mathbf{M}$ cannot be deemed as incorrect until its constituent triples are deemed to belong to $\mathbf{M}$. This leads to a recursive check of all procedures' subprocedures to the lowest levels of abstraction. Thus, the ability to localize a fault by considering only the structures at a given level of abstraction is lost. Without dependency constraints at each level of abstraction as utilized by the expectations models of violated expectations, one is forced to consider the structures at *all* levels of abstraction.

The label of incorrectness is based on a membership test of a triple in a set of triples. Consider a triple from the dynamic model of the artifact and what might make it differ from all triples in $\mathbf{M}$:

- a different procedure name implies that perhaps the artifact procedure name is misspelled or that the procedure has no place being invoked in any expected computation of the program;
- different data variable values implies that some computation of the procedure not considered in the model (*i.e.* not computed by a procedure call) is faulty or that the order of variables in the input or output vectors differ from what is expected⁴;
- extra/missing data variables from the input or output vectors implies some inconsistency between the interstructural dependencies local to the procedure of the artifact and those expected of the artifact.

⁴The Prolog language also loosely *types* its data variables. Thus, differences in variable types can also be grounds for incorrectness.

The fault may, in fact, be due to any or all of these factors.

Since interstructural dependencies data variable dependencies are not accounted for, the identity of procedures which influence or are dependent on the values of variables local to a procedure is not a criterion for correctness. Since interstructural event variable dependencies are not accounted for, the *order* of procedure invocation is not a criterion for correctness either.

6.2.4 Termination With Missing Output

A *missing output error* can occur in a nondeterministically modelled system. For a nondeterministic language like Prolog “it may happen that every computation of p on x terminates and returns an output correct in M , but no computation returns y ” [Shapiro84]. In Prolog, this kind of error is referred to as *finite failure*. Shapiro attributes such an error to the existence of an *incomplete procedure* in the artifact. Either an incomplete procedure is one with the constituent procedures in the dynamic artifact model, at least one of which is unexpected or the incomplete procedure is an unexpected procedure which is not observed in the dynamic artifact model.

Here the diagnostic process is interleaved with the generation of a dynamic causal model of *expectations* from the static causal model of the *artifact* using the input and missing output values as initial conditions. The diagnostic process controls the evaluation mechanism as follows: the unexpected triple is tested for finite failure by letting the evaluation mechanism (the program interpreter) determine that the triple is inconsistent with all possible dynamic models which can be generated from the static model (the faulty program). This done, the diagnostic procedure chooses the child procedures which constitute the faulty triple in one of the inconsistent dynamic models. The diagnostic procedure first queries an existential oracle to determine whether these constituent triples belong to any expected dynamic model. If any do not then the parent procedure is deemed incomplete. Otherwise the diagnostic procedure queries the existential oracle for all possible values of uninstantiated data variables that the constituent procedures might be expected to take on in all possible dynamic models of expectations. The diagnostic procedure then proceeds to recursively consider each of the instantiated triples as it did the parent triple. An

example of this procedure is included in Appendix A.

The procedure is rather query intensive. Pereira [Pereira86] improves the procedure somewhat by only querying the oracle regarding whether a goal is expected to fail or not freeing the oracle from having to provide values for data variables.

This diagnostic process is similar to that utilized for violated expectations. The violated expectations approach detects conflicts by generating a dynamic causal model from a static causal model of *expectations* using initial conditions observed in the *artifact*. PDS6 detects inconsistencies identifying an incomplete procedure by generating a dynamic causal model from a static causal model of the *artifact* using initial conditions from *expectations*. This observation leads to the conclusion that a missing solution in the artifact can, in fact, be interpreted as an incorrect solution in expectations.

Where does the fault lie for a procedure designated as incomplete? Either one or more of its instantiated subprocedures may not belong to $\mathbf{M}$ or the instantiated parent triple which is in $\mathbf{M}$ is inconsistent (*immediately fails*) with any dynamic model of the artifact. Either situation might be due to any of the reasons a procedure may be designated as incorrect as discussed in the last section. However, in the case of differing data variable values, one could infer that *any* computation of the procedure (*i.e.* including computations invoked by constituent procedure calls) is incorrect.

6.2.5 Nontermination Error

A nontermination error occurs when a program either “exhausts the space allocated to running it (and on any existing machine this amount has some fixed upper bound), or the patience of the person waiting for its output (a quantity not proven to be bounded, but in the following assumed to be so)” [Shapiro84]. Indeed, a program is not necessarily nonterminating if it has simply exhausted one of the above resources. For this reason the diagnosis algorithm for nontermination may fail to locate any fault. However, when it does locate a fault, it is due to an *incorrect procedure* as already described or due to a *procedure that diverges with respect to $\succ$ and $\mathbf{M}$* .

The latter fault is defined to be a triple $\langle \mathbf{p}, x, y \rangle$ from the dynamic artifact model with a constituent triple $\langle \mathbf{q}, u, v \rangle$ which violates the well founded ordering

such that $\langle p, x \rangle \not\succ \langle q, u \rangle$ and all constituent triples in the dynamic model which causally precede $\langle q, u, v \rangle$ are expected (*i.e.* in $\mathbf{M}$).

The diagnostic procedure examines consecutive procedures $\langle p, x \rangle$ and $\langle q, u \rangle$ on the call stack of the artifact and queries an $\succ$ oracle to check whether they violate the $\succ$ ordering. If none violate the ordering then the diagnostic procedure terminates having found no fault. If a violation is detected, the diagnostic procedure compares procedures called by p before q is called with those expected by querying a ground oracle. If one of these constituent procedures is found to return an unexpected output then the incorrect solution diagnostic procedure is invoked. Otherwise the nontermination diagnostic procedure identifies p as the diverging procedure.

Note that this is the only PDS6 diagnostic procedure which utilizes information concerning expected and actual interstructural event variable dependencies; that is to say, the expected and actual causal paths taken in the respective causal models.

6.2.6 Incorrect Solution *vs.* Missing Solution

Given appropriate models of an artifact and expectations, Shapiro's incorrect solution and missing solution diagnostic procedures are applicable to the diagnosis of missing and incorrect solutions respectively. In fact, a procedure similar to that of the missing solution procedure is used in violated expectations for the diagnosis of incorrect solutions in hardware.

As discussed in §6.2.4 an incorrect solution in an artifact model can be interpreted as a solution missing from the expectations model. To locate the incomplete procedure for a missing solution error a static causal model of expectations is required. In the diagnostic domain of PDS6 this is equivalent to a corrected copy of the faulty program – a copy of which would render the diagnostic process pointless. When diagnosing faults in hardware however, a detailed static causal model of expectations is sometimes available in the form of a detailed design specification or simulation model.

Considering the diagnostic procedure for locating incorrect solutions, a solution missing from the artifact model can be interpreted as an incorrect solution in the expectations model. Again we require a correct copy of the faulty program, this

time as a static causal model of the artifact for it is in our model of expectations (the correct program) in which we have ‘observed’ an incorrect solution. Though realistically inapplicable in the domain of PDS6 this interpretation of the incorrect solution diagnostic procedure suggests a procedure for diagnosing missing solutions in hardware.

As examples of the two procedures’ duality we have included in the Appendix Shapiro’s incorrect solution example reinterpreted as a missing solution and his missing solution example reinterpreted as an incorrect solution.

6.3 Applicability to Our Objectives

We now consider the applicability of these diagnostic procedures to the models of concurrent software outlined in Chapter 5.

6.3.1 Nontermination

All the procedures here assume the dynamic models are complete in the sense that all input variables applied to the artifact which influence an erroneous output variable are accounted for. For Shapiro’s procedures this assumption takes the form of assuming program termination and enough memory to store the entire computation tree during the evaluation process. As discussed in Chapter 5, the nonterminating (*not* in a diverging sense) nature of concurrent software makes it impossible to obtain a complete dynamic artifact model. Thus, an analysis of the incomplete model must occur to identify which, if any, unaccounted for input variables exist for a given error so appropriate action may be taken.

6.3.2 Nondeterminism

In Chapter 5 we determined that nondeterminism may be encountered during the construction of dynamic artifact models. Shapiro’s missing solution procedure accommodates nondeterminism of the artifact model by querying an existential oracle and utilizing an evaluation mechanism which considers all possible execution paths.

This seems like a good approach but Shapiro's procedure is applicable only to terminating programs, an unlikely occurrence for concurrent software applications.

6.3.3 Intrastructural Dependency Constraints

Since the static artifact model for concurrent software does not include constraints on intrastructural data variables dependencies the violated expectations technique is not applicable for finding faults associated with incorrectly valued data variables except at the lowest level of structural abstraction. Rather than isolating faults by simply considering a single level of the structural hierarchy, a procedure must consider *all* level of the structural hierarchy as Shapiro's missing and incorrect solution procedures do.

6.3.4 Refining the Search Space

One important unexplored avenue of research is determining the characteristic size of the search space for fault diagnosis in the concurrent software domain. The performance of PDS6 and violated expectations on large problems hasn't been analyzed in the literature. Taking into account interstructural data variable dependencies as practiced by violated expectations and Pereira helps to refine the space beyond the space defined by PDS6. Taking into account event variable dependencies and utilizing a half-split search strategy or some suitable heuristic should further refine the space.

6.3.5 Model Detail

The diagnostic procedures employed should be able to accommodate the varying levels of detail in dynamic models. For example, the model resulting from hardware monitoring initially consists of event variables. We must determine how a procedure might diagnose an incorrectly valued data variable using this model.

6.3.6 Fault Coverage

Violated expectations has shown itself to locate faults due to inconsistent intrastructural data-event variable constraints. Shapiro's procedures locate three types of fault all of which are quite ambiguous regarding the specific location of the fault. Neither violated expectations or PDS6 locates faults due to incorrectly constrained event variable dependencies. We would like to refine the definitions for faults occurring in concurrent software so that a fault can be more precisely located.

6.4 Summary

The process of behavioral diagnosis is quite straightforward given complete dynamic causal models of the artifact and expectations. Complications arise when the only models available are incomplete, nondeterministic or insufficiently detailed. We have examined two diagnostic approaches which attempt to deal with different model characteristics in order to better understand what sort of procedure might be utilized in the concurrent software domain. Development of a suitable procedure cannot proceed until some example artifact and expectations models are constructed from the concurrent software domain to precisely determine their characteristics.

Chapter 7

Conclusions and Future Research

7.1 A Research Summary

Debugging concurrent software is currently far more of an art than a science. Using traditional debugging tools a diagnostician can quickly become overwhelmed by the amount of information available for the interpretation of a bug. The goal of this thesis has been to examine the utilization of knowledge-based diagnosis in an effort to assist the diagnostician in the task of locating faults in concurrent software. In particular, we have focused on locating faults manifesting semantic errors during the implementation phase of software development. The implementation environment of interest consists of a uniprocessor host execution environment and a multiprocessor target execution environment.

A summary of our research results follows.

7.1.1 Diagnostic Taxonomies

We have proposed taxonomies of diagnostic processes and diagnostic models which provide bases for a critical examination of their effectiveness in diagnostic systems. We have found that no particular diagnostic process coupled with a particular diagnostic model is most effective for any given diagnostic problem. The choice of a suitable process and model must be made considering the associated diagnostic domain. We have concluded that for the concurrent software domain a *behavioral diagnostic process* combined with *deep causal models* of expectations and the faulty

program is most suitable.

7.1.2 Deep Causal Modelling

Having found the traditional deep causal modelling formalism to be lacking in its representational abilities for concurrent software we have proposed a number of extensions to the formalism. Our extended modelling formalism accommodates many of the characteristics of concurrent software including synchronization and communication between tasks, conditional execution of code and abstraction of code and data. We leave the modelling of dynamic installation/removal of code and data and indirect reference of code and data to future research.

Within the modelling framework we have proposed comprehensive definitions of faults and errors and discuss how errors defined in this manner might be observed in an executing program.

7.1.3 Model Sources

We have identified four alternative sources of models in our stated implementation environment. Model construction utilizes existing tools including the host machine symbolic debugger, the concurrent software execution simulator and the target machine symbolic debugger. We have also identified the need for new tools including a concurrent program dependence graph generator, a target machine execution replay tool and a target machine monitor implemented as a hardware subsystem for the target.

Having compared their relative advantages and disadvantages we have presented a possible progression for utilizing the alternative sources which corresponds to the phases of concurrent software implementation; namely, simulation on the host followed by execution on the target.

7.1.4 Diagnostic Processes

We have examined two existing approaches to behavioral diagnosis utilizing deep causal models; one intended for the hardware domain and one intended for the diagnosis of sequential Prolog programs. Our examination has revealed an interesting

duality between the two approaches.

Although we do not go as far as proposing a diagnostic process of our own we do present an evaluation of the two approaches with respect to our needs.

7.2 Assessment of Results

The results of our research have succeeded in identifying the form a knowledge-based diagnostic system might take in an implementation environment for concurrent software applications. Many aspects associated with such a system have been identified including a badly needed clarification of terminology used to discuss elements of the system and the proposal of a modelling formalism capable of representing the diagnostic domain.

7.3 Future Research

The problem of knowledge-based diagnosis for concurrent software still requires further refinement before a diagnostic system can be implemented. To be a practical tool the modelling formalism must be able to represent dynamic creation/destruction of code structures and data. It must also be able to represent pointer operations. A suitable diagnostic process must be developed to infer from and compare models of the software artifact and expectations. The entire system should be formalized to make it possible to prove properties of the system regarding fault coverage and computational complexity.

An implementation would serve to confirm the usefulness of the system. Observations of such a system would help to characterize the nature of the concurrent software problem space encountered by the diagnostic system. The results of this research could take the form of optimizations in the system.

We conclude by presenting a high level structural overview of a diagnostic system based on the results of our investigation into possible diagnostic modelling sources for concurrent software. The structural decomposition for this system is shown in Figure 7.1.

Figure 7.1: High level structural decomposition of the diagnostic system

7.3.1 Diagnostic System

The overall function of the diagnostic system is to attempt to identify the location of program faults which have caused errors reported to the system by the diagnostician. During the course of fault location the system is expected to interact with the diagnostician.

7.3.2 External Environment

The environment external to the diagnostic system consists of an I/O module which interfaces to the host computer, a file module which provides access to the application code and symbol table through the host file manager, an interface to the monitor subsystem resident on the target, the Melody target system debugger and the CopyCat/dbxtool host system debugger.

7.3.3 Expectations Model Manager

The expectations model manager is responsible for constructing a *consistent* causal model of static and/or dynamic system expectations. It does this by posing queries to the diagnostician only those queries *necessary* for the diagnostic task at hand. In an attempt to make the queries *comprehensible* they may be accompanied by simple graphical depictions of program flow akin to the chosen modelling formalism.

7.3.4 Artifact Model Manager

The artifact model manager is responsible for constructing a static and/or dynamic causal model of the artifact. The static model is the result of transforming the application software into a program dependence graph. The dynamic model is based on information from one of these sources:

- information from the CopyCat/dbxtool host system debugger combination;
- information from a replay mechanism/Melody target system debugger combination;
- information from the Melody target system debugger trace buffers;
- information from the event stream recorded by a target hardware monitor.

Any missing details of the model are added through analysis of the program dependence graph, symbol table and by consultation with the diagnostician.

7.3.5 User Interface

The user interface provides a consistent format for queries to the diagnostician and replies from the diagnostician. It should support both graphical and textual I/O.

7.3.6 Diagnosis Manager

The diagnosis manager compares aspects of the artifact and expectations models in response to errors submitted by the diagnostician. It queries the artifact model manager, the expectations model manager and the diagnostician during the course

of the diagnosis. The diagnosis manager should provide the flexibility to alter diagnostic strategies, the ability to alter replies to queries and keep the diagnostician informed of its progress.

References

- [Adam80] Anne Adam, Jean-Pierre Laurent, "LAURA, A System to Debug Student Programs," *Artificial Intelligence*, vol. 15, no. 1,2, 1980, pp. 75-122.
- [Aho86] Alfred V. Aho, R. Sethi, Jefferey D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.
- [ALRM83] U.S. Department of Defence, *Reference Manual for the Ada Programming Language MIL-STD-1815a*, U.S. Department of Defence, 1983.
- [Baiardi86] Fabrizio Baiardi, Nicoletta De Francesco, Gigliola Vaglini, "Development of a Debugger for a Concurrent Language," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 4, April 1986, pp. 547-553.
- [Bates88] Peter Bates, "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior," *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, ACM, Madison, WI, May 1988, pp. 11-22.
- [Bobrow84] Daniel G. Bobrow, "Qualitative Reasoning About Physical Systems: An Introduction," *Artificial Intelligence*, vol. 24, 1984, pp. 1-5
- [Bobrow85] Daniel G. Bobrow ed., *Qualitative Reasoning About Physical Systems*, MIT Press, Cambridge, MA, 1985.
- [Boyer75] Robert S. Boyer, J. Strother Moore, "Proving Theorems About LISP Functions," *Journal of the ACM*, vol. 22, no. 1, January 1975, pp. 129-144.
- [Bruegge83] Bernd Bruegge, Peter Hibbard, "Generalized Path Expressions: A High Level Debugging Mechanism," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging*, Mark S. Johnson (ed.), ACM, Pacific Grove, CA, March 1983, pp. 34-44.

- [Buhr88] Raymond J. A. Buhr, "Machine Charts for Visual Prototyping in System Design," *SCE Technical Report 88-2*, Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, August 1988.
- [Cheung87] W.H. Cheung, James P. Black, Eric G. Manning, "A Study of Distributed Debugging," *Research Report CS-87-53*, Faculty of Mathematics, University of Waterloo, Waterloo, ON, October 1987.
- [Davis82a] R. Davis, H. Shrobe, W. Hamscher, K. Wieckert, M. Shirley, S. Polit, "Diagnosis Based on a Description of Structure and Function," *Proceedings of the National Conference on Artificial Intelligence*, AAAI-82, Pittsburgh, PA, Aug. 1982, pp. 137-142.
- [Davis82b] Randall Davis, "Expert Systems: Where Are We? And Where Do We Go From Here?," *The AI Magazine*, vol. 3, no. 2, Spring 1982, pp. 3-22.
- [Davis84] Randall Davis, "Diagnostic Reasoning Based on Structure and Behavior," *Artificial Intelligence*, vol. 24, 1984, pp. 347 - 410.
- [deKleer76] Johan de Kleer, "Local Methods for Localizing Faults in Electronic Circuits," Artificial Intelligence Laboratory, AI Memo No. 394, MIT, Cambridge, MA, 1976.
- [deKleer77] Johan de Kleer, "Multiple Representations of Knowledge in a Mechanics Problem-Solver," *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, IJCAI-77, 1977, pp. 299-304.
- [deKleer79] Johan de Kleer, "The Origin and Resolution of Ambiguities in Causal Arguments," *Proceedings of the 6th International Joint Conference on Artificial Intelligence*, IJCAI-79, 1979, pp. 197-203.
- [deKleer87] Johan de Kleer, Brian C. Williams, "Diagnosing Multiple Faults," *Artificial Intelligence*, vol. 32, 1987, pp. 97-130.
- [Ducasse88] Mirielle Ducassé, Anna-Marie Emde, "A Review of Automated Debugging Systems: Knowledge, Strategies and Techniques," *10th International Conference on Software Engineering*, Singapore, April 1988, pp. 162-171.
- [DVME-134] DY-4 Systems, *DVME-134 Single Board Computer Operations Manual*, DY-4 Systems Inc., Nepean, ON, March 1988.

- [Feldman88] Stuart I. Feldman, Channing B. Brown, "IGOR: A System for Program Debugging via Reversible Execution," *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, ACM, Madison, WI, May 1988, pp. 112-123.
- [Finger85] J.J. Finger, Michael R. Genesereth, "RESIDUE: A Deductive Approach to Design Synthesis," *Stanford Heuristic Programming Project, Memo HPP-85-1*, Computer Science Department, Stanford University, Stanford, CA, 1985.
- [Fischhoff79] Baruch Fischhoff, "Diagnosing Clinical Diagnosis", *Contemporary Psychology*, vol. 24, no. 1, January 1979, pp.48-49.
- [Forbus81] Kenneth D. Forbus, "Qualitative Reasoning About Physical Processes," *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, IJCAI-81, 1981, pp. 326-330.
- [Forbus84] Kenneth D. Forbus, "Qualitative Process Theory," *Artificial Intelligence*, vol. 24, 1984, pp. 85-168.
- [Fosdick76] Lloyd D. Fosdick, Leon J. Osterweill, "Data Flow Analysis In Software Reliability," *ACM Computing Surveys*, vol. 8, no. 3, September 1976, pp. 305-330.
- [Genesereth84] Michael R. Genesereth, "The Use of Design Descriptions in Automated Diagnosis," *Artificial Intelligence*, vol. 24, 1984, pp. 411-436.
- [Gould75] John D. Gould, "Some Psychological Evidence on How People Debug Computer Programs," *International Journal of Man-Machine Studies*, vol. 7, 1975, pp. 151-182.
- [Gupta84] N.K. Gupta, R.E. Seviora, "An Expert System Approach to Real-Time System Debugging," *First Conference on Artificial Intelligence Applications*, IEEE/AAAI, Denver, CO., December 1984, pp. 336-343.
- [Hamscher84] Walter Hamscher, Randall Davis, "Diagnosing Circuits with State: An Inherently Underconstrained Problem," *Proceedings of the National Conference on Artificial Intelligence*, AAAI-84, Austin, TX, 1984, pp. 142 - 147.
- [Harandi83] Mehdi T. Harandi, "Knowledge-Based Program Debugging: A Heuristic Model," *Proceedings of the 1983 Softfair - Software Development: Tools, Techniques and Alternatives*, CS Press, Los Alamos, CA, 1983, pp. 282-288.

- [Hayes79] Patrick J. Hayes, "The Naive Physics Manifesto," D. Michie (Ed.), *Expert Systems in the Electronic Age*, Edinburgh University Press, Edinburgh, 1979, pp. 242-270.
- [Helmbold85] David Helmbold, David Luckham, "Debugging Ada Tasking Programs," *IEEE Software*, vol. 2, no. 2, pp. 47-57, March 1985.
- [Helmbold88] David Paul Helmbold, "The Meaning of TSL: An Abstract Implementation of TSL-1," *Technical Report No. UCSC-CRL-87-29*, Computer Research Laboratory, University of California, Santa Cruz, CA, March 1988.
- [Hogger84] Christopher John Hogger, "Introduction to Logic Programming," Academic Press Inc., London, 1984.
- [Hudlicka88] Eva Hudlická, "Construction and Use of a Causal Model for Diagnosis," *International Journal of Intelligent Systems*, vol. 3, 1988, pp. 315-349.
- [Hudlicka86] Eva Hudlická, "Diagnosing Problem-Solving Behavior," Ph.D. Dissertation, Department of Computer and Information Science, University of Massachusetts, February 1986.
- [Iwasaki86] Yumi Iwasaki, Herbert A. Simon, "Causality in Device Behavior," *Artificial Intelligence*, Vol. 29, 1986, pp. 3-32.
- [Johnson84] W.L. Johnson, E. Soloway, "Intention-Based Diagnosis of Programming Errors," *Proceedings of the National Conference on Artificial Intelligence*, AAAI-84, Austin, Texas, 1984, pp. 162-168.
- [Johnson86] S.C. Johnson, "Lint, a C Program Checker," *UNIX Programmers Manual - Supplementary Documents, Vol. 1 (PS1)*, Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkley, CA, April 1986, pp. PS1:9-1 - PS1:9-11.
- [King76] James C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, no. 7, July 1976, pp. 385-394.
- [Korel86] Bogdan Korel, "A Program Error Localization Expert System," *SPIE Proceedings - Applications of Artificial Intelligence III*, vol. 635, 1986, pp. 111-117.
- [Kuipers84] Benjamin Kuipers, "Commonsense Reasoning about Causality: Deriving Behavior from Structure," *Artificial Intelligence*, vol. 24, 1984, pp. 169-203.

- [LeBlanc87] T.J. LeBlanc, J.M. Mellor-Crummey, "Debugging using Instant Replay," *IEEE Transactions on Computers*, vol. C-36, no. 4, April 1987, pp. 471-482.
- [LeDoux85] Carol Helfgott LeDoux, "A Knowledge-Based System For Debugging Concurrent Software," Ph.D. Thesis, Department of Computer Science, University of California, Los Angeles, CA, 1985.
- [Luckham86] David C. Luckham, Randall Neff, David Rosenblum, "An Environment for Ada Software Development Based on Formal Specification," *Technical Report No. CSL-TR-86-305*, Program Analysis and Verification Group Report No. 31, Stanford University, Stanford, CA, August 1986.
- [Lukey80] F.J. Lukey, "Understanding and Debugging Programs," *International Journal of Man-Machine Studies*, vol. 12, no. 2, February 1980, pp. 189-202.
- [Lyle87] James R. Lyle, Mark Weiser, "Automatic Program Bug Location by Program Slicing," *The 2nd International Conference on Computers and Applications*, IEEE, Beijing, June 1987, pp. 877-883.
- [McDowell87] Charles E. McDowell, David P. Helmbold, Anil K. Sahai, "A Survey of Debugging Tools for Concurrent Programs," *Technical Report UCSC-CRL-87-22*, Computer Research Laboratory, University of California, Santa Cruz, CA, December 1987.
- [Michie81] Donald Michie, "High-Road and Low-Road Programs," *AI Magazine*, vol. 3, no. 1, Winter 1981-82, pp. 21-22.
- [Miller88] Barton P. Miller, Jong-Deok Choi, "A Mechanism for Efficient Debugging of Parallel Programs," *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, ACM, Madison, WI, May 1988, pp. 141-150.
- [Murray87] William R. Murray, "Automatic Program Debugging for Intelligent Tutoring Systems," *National Research Council Computational Intelligence*, vol. 3, no. 1, February 1987, pp. 1-16.
- [Nelson82] William R. Nelson, "Reactor: An Expert System For Diagnosis and Treatment of Nuclear Reactor Accidents", *Proceedings of the National Conference on Artificial Intelligence*, AAAI-82, Pittsburgh, PA, August 1982, pp. 296-301.

- [Patil81] Ramesh S. Patil, Peter Szolovits, William B. Schwartz, "Causal Understanding of Patient Illness in Medical Diagnosis," *Proceedings of the 7th International Joint Conference on Artificial Intelligence, IJCAI-81*, vol. 2, 1981, pp. 893-899.
- [Pereira86] Luis Moniz Pereira, "Rational Debugging in Logic Programming," *Proceedings of the Third International Conference on Logic Programming*, Ehud Shapiro (ed.), London, July 1986, Springer-Verlag, pp. 203-210.
- [Reiter87] Raymond Reiter, "A Theory of Diagnosis from First Principles," *Artificial Intelligence*, vol. 32, 1987, pp. 57-95.
- [Rich81] Charles Rich, "A Formal Representation for Plans in the Programmer's Apprentice," *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, Los Altos, CA, 1981, pp. 1044-1052.
- [Rich88] Charles Rich, Richard C. Waters, "The Programmer's Apprentice: A Research Overview," *IEEE Computer*, vol. 21, no. 11, November 1988, pp. 11-25.
- [Rockstrom82] Anders Rockström, Roberto Saracco, "SDL – CCITT Specification and Description Language," *IEEE Transactions on Communications*, vol. COM-30, no. 6, June 1982, pp. 1310-1318.
- [Rosenblum86] David Rosenblum, Sriram Sankar, David C. Luckham, "Concurrent Runtime Checking of Annotated Ada Programs," *Technical Report No. CSL-TR-86-312*, Program Analysis and Verification Group Report No. 33, Stanford University, Stanford, CA, November 1986.
- [Schwartz71] Jacob T. Schwartz, "An Overview of Bugs," *Courant Computer Science Symposium – Debugging Techniques in Large Systems*, Randall Rustin (ed.), Prentice-Hall, 1971, pp. 1-16.
- [Schwartz83] Richard L. Schwartz, P.M. Melliar-Smith, Friedrich H. Vogt, "An Interval-Based Temporal Logic," *Proceedings of Logics of Programs – Lecture Notes in Computer Science Vol. 164*, E. Clarke and D. Kozen (ed.), Springer-Verlag, New York, 1983, pp. 443-457.
- [Sedlmeyer83] Robert L. Sedlmeyer, William B. Thompson, Paul E. Johnson, "Knowledge-Based Fault Localization in Debugging," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging*, M.S. Johnson (ed.), ACM, Pacific Grove, CA, March 1983, pp. 25-31

- [Seidner83] Rich Seidner, Nick Tindall, "Interactive Debugging Requirements," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging*, Mark S. Johnson (ed.), ACM, Pacific Grove, CA, March 1983, pp. 9-22.
- [Seviora87] Rudolph E. Seviora, "Knowledge-Based Program Debugging Systems," *IEEE Software*, vol. 4, no. 3, May 1987, pp. 20-32.
- [Shapiro81] Daniel G. Shapiro, "Sniffer: a System that Understands Bugs," AI Memo No. 638, MIT, Cambridge, MA, June 1981.
- [Shapiro84] Ehud Shapiro, "Algorithmic Program Debugging," The MIT Press, Cambridge, MA, 1984.
- [Sterling86] Leon Sterling, Ehud Shapiro, *The Art of Prolog*, MIT Press, 1986.
- [Shirley83] M.H. Shirley, Randall Davis, "Generating Distinguishing Tests Based On Hierarchical Models and Symptom Information," *Proceedings IEEE International Conference on Computer Design*, Rye, NY, 1983.
- [Shortcliffe76] Edward H. Shortcliffe, *Computer Based Medical Consultations: MYCIN*, Elsevier, North Holland, 1976.
- [Statemate87] i-Logix, *The Languages of Statemate*, i-Logix Inc., November 1987.
- [Sussman80] Gerald J. Sussman, Guy L. Steele Jr., "CONSTRAINTS - A Language For Expressing Almost-Hierarchical Descriptions," *Artificial Intelligence*, vol. 14, 1980, pp. 1-39.
- [Taylor80] Richard N. Taylor, Leon J. Osterweil, "Anomaly Detection in Concurrent Software by Static Data Flow Analysis," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 3, May 1980, pp. 265-278.
- [Taylor83] Richard N. Taylor, "A General-Purpose Algorithm for Analyzing Concurrent Programs," *Communications of the ACM*, vol. 26, no. 5, May 1983, pp. 362-376.
- [Tischler83] Ron Tischler, Robin Schaufler, Charlotte Payne, "Static Analysis as an Aid to Program Debugging," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging*, Mark S. Johnson (ed.), ACM, Pacific Grove, CA, March 1983, pp. 155-158.
- [Unison87a] Multiprocessor Toolsmiths, *Unison Real-Time Multitasking, Multiprocessing Operating System User's Guide*, Multiprocessor Toolsmiths Inc., Kanata, ON, August 1987.

- [Unison87b] Multiprocessor Toolsmiths, *Unison Support Tools – CopyCat, Remedy Debugger*, Multiprocessor Toolsmiths Inc., Kanata, ON, August 1987.
- [Weiser82] Mark Weiser, "Programmers Use Slices When Debugging", *Communications of the A.C.M.*, vol. 25, no. 7, July 1982, pp. 446-452.
- [Weiss78] Sholom M. Weiss, Casimir A. Kulikowski, Saul Amarel, Aran Safir, "A Model-Based Method for Computer-Aided Medical Decision Making," *Artificial Intelligence*, vol. 11, pp. 145-172, 1978.
- [Wertz82] Harold Wertz, "Stereotyped Program Debugging: An Aid for Novice Programmers," *International Journal of Man-Machine Studies*, vol. 16, 1982, pp. 379-392.
- [Wittie88] Larry D. Wittie, "Debugging Distributed C Programs by Real Time Replay," *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, ACM, Madison, WI, May 1988, pp. 57-67.
- [Wybranietz88] Dieter Wybranietz, Dieter Haban, "Monitoring and Performance Measuring Distributed Systems During Operation, " *ACM*, 1988, pp. 197-206.

Appendix A

Reinterpretation of Shapiro's Diagnostic Procedures

A.1 Incorrect Solution as a Missing Solution

As discussed in §6.2.6, Shapiro's example for the diagnosis of an incorrect solution can be interpreted as the diagnosis of a missing solution. Figure A.1 depicts a faulty insertion sort program in Prolog used as the artifact for Shapiro's incorrect solution example. Figure A.2 depicts a corrected version of the program; the arguments to the inequality of the first clause of the *insert* procedure being reversed from that in the artifact. Figures A.3 and A.4 respectively depict proof trees of the faulty program artifact and the corrected program (reflecting expectations) for the same input values.

Figure A.5 reproduces the dialogue between Shapiro's incorrect solution diagnostic procedure *fp* and the diagnostician. The diagnostician's dialogue is in boldface. Recall that this procedure utilizes the faulty program as a static causal model with initial conditions being observed input and faulty output variable values. Queries to the diagnostician (ground oracle) determine whether a fully instantiated procedure upon returning from a call is consistent with expectations. The last query in the dialogue establishes that *insert*(1,[3],[3,1]) is inconsistent. The clause responsible for the instantiation is returned implying that *insert* is the incorrect procedure responsible for the observed incorrect solution error.

$isort([X|X_s], Y_s) \leftarrow isort(X_s, Z_s), insert(X, Z_s, Y_s).$
 $isort([], []).$

 $insert(X, [Y|Y_s], [Y|Z_s]) \leftarrow Y > X, insert(X, Y_s, Z_s).$
 $insert(X, [Y|Y_s], [X, Y|Y_s]) \leftarrow X \leq Y.$
 $insert(X, [], [X]).$

Figure A.1: Static Causal Artifact Model

$isort([X|X_s], Y_s) \leftarrow isort(X_s, Z_s), insert(X, Z_s, Y_s).$
 $isort([], []).$

 $insert(X, [Y|Y_s], [Y|Z_s]) \leftarrow X > Y, insert(X, Y_s, Z_s).$
 $insert(X, [Y|Y_s], [X, Y|Y_s]) \leftarrow X \leq Y.$
 $insert(X, [], [X]).$

Figure A.2: Static Causal Model of Expectations

Figure A.3: Dynamic Causal Artifact Model

Figure A.4: Dynamic Causal Model of Expectations

$|\text{?} - \text{fp}(\text{isort}([2, 1, 3], [2, 3, 1]), C).$

query : $\text{isort}([], [])?$ **y**.

query : $\text{insert}(3, [], [3])?$ **y**.

query : $\text{isort}([3], [3])?$ **y**.

query : $\text{insert}(1, [], [1])?$ **y**.

query : $\text{insert}(1, [3], [3, 1])?$ **n**.

$C = \text{insert}(1, [3], [3, 1]) \leftarrow 3 > 1, \text{insert}(1, [], [1])$

yes

Figure A.5: Dialogue for Incorrect Solution

```

|? – ip(isort([2, 1, 3], [2, 3, 1]), C).

query : isort([1, 3], X)? y.
which X? [3, 1].

query : insert(2, [3, 1], [2, 3, 1])? y.

query : isort([3], X)? y.
which X? [3].

query : insert(1, [3], [3, 1])? y.

C = insert(1, [3], [3, 1])

yes

```

Figure A.6: Dialogue for Missing Solution

Interpreting the incorrect solution in the artifact as a missing solution in expectations requires that we use the correct program as a static causal artifact model with the missing input/output variable value pair serving as initial conditions. Furthermore, the faulty program and all of its proof trees are treated as dynamic causal models of expectations. Figure A.6 shows the dialogue between Shapiro’s missing solution diagnostic procedure *ip* and the diagnostician. Queries to the diagnostician (an existential oracle) determine whether a procedure call (instantiated or uninstantiated) is consistent with expectations. If so, all expected values for uninstantiated variables are prompted for. In this case *isort* is a deterministic procedure so only one value is necessary for an uninstantiated variable. The last query in the dialogue establishes that *insert*(1, [3], [1, 3]) is expected (from the faulty program) but is inconsistent with the artifact (the correct program). The instantiated expected procedure is returned implying that *insert* is the incomplete procedure responsible for the observed missing solution error.

$$\begin{aligned}
&isort([X|X_s], Y_s) \leftarrow isort(X_s, Z_s), insert(X, Z_s, Y_s). \\
&isort([], []). \\
\\
&insert(X, [Y|Y_s], [Y|Z_s]) \leftarrow X > Y, insert(X, Y_s, Z_s). \\
&insert(X, [Y|Y_s], [X, Y|Y_s]) \leftarrow X \leq Y.
\end{aligned}$$

Figure A.7: Static Causal Artifact Model

Figure A.8: Dynamic Causal Model of Expectations

A.2 Missing Solution as an Incorrect Solution

Figure A.7 depicts a faulty insertion sort program in Prolog used as the artifact for Shapiro's missing solution example. Figure A.2 depicts the correct version of the program; a third clause being added to the *insert* procedure. Figure A.8 depicts the proof tree of the correct program for the observed input and missing output values.

Figure A.9 reproduces the dialogue between Shapiro's missing solution diagnostic procedure *ip* and the diagnostician. Recall that this procedure utilizes the faulty program as a static causal model with initial conditions being the missing input/output variable value pair. The last query in the dialogue establishes that *insert*(1, [], [1]) is expected (from the correct program) but is inconsistent with the artifact (the incorrect program). The instantiated expected procedure is returned implying that *insert* is the incomplete procedure responsible for the observed missing solution error.

$|\text{?} - \text{ip}(\text{isort}([\mathbf{3}, \mathbf{2}, \mathbf{1}], [\mathbf{1}, \mathbf{2}, \mathbf{3}]), \mathbf{X}).$

query : *isort*([2, 1], *X*)? **y**.

which X? **[1, 2]**.

query : *insert*(3, [1, 2], [1, 2, 3])? **y**.

query : *isort*([1], *X*)? **y**.

which X? **[1]**.

query : *isort*([], *X*)? **y**.

which X? **[]**.

query : *insert*(1, [], [1])? **y**.

X = *insert*(1, [], [1])

yes

Figure A.9: Dialogue for Missing Solution

$! ? - \text{fp}(\text{isort}([3, 2, 1], [1, 2, 3]), X).$

query : *isort*([1], [1])? *y*.

query : *insert*(2, [], [2])? *n*.

$X = \text{insert}(2, [], [2])$

yes

Figure A.10: Dialogue for Incorrect Solution

Interpreting the missing solution from the artifact as an incorrect solution in expectations requires that we use the correct program as a static causal artifact model with the incorrect input/output value pair serving as initial conditions. Furthermore, the faulty program and all of its proof trees are treated as dynamic causal models of expectations. Figure A.10 shows the dialogue between Shapiro's incorrect solution diagnostic procedure *fp* and the diagnostician. The last query in the dialogue establishes that *insert*(2, [], [2]) (from the correct program) is inconsistent with expectations (the faulty program). The clause responsible for the instantiation is returned implying that *insert* is the incorrect procedure responsible for the observed incorrect solution error.

END

1 8·0 2·9 0

FIN