Detecting and ~~Locating~~ Correcting Errors in Parallel Object Oriented Systems

Jan A. Purchase

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
of the
University of London

Department of Computer Science
University College London

December 1991

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Abstract

Our research concerns the development of an operational formalism for the in-source specification of parallel, object oriented systems. These specifications are used to enunciate the behavioural semantics of objects, as a means of enhancing their reliability.

A review of object oriented languages concludes that the advance in language sophistication heralded by the object oriented paradigm has, so far, failed to produce a commensurate increase in software reliability. The lack of support in modern object oriented languages for the notion of 'valid object behaviour', as distinct from state and operations, undermines the potential power of the abstraction. Furthermore, it weakens the ability of such languages to detect behavioural problems, manifest at run-time. As a result, in-language facilities for the signalling and handling of undesirable program behaviours or states (for example, assertions) are still in their infancy. This is especially true of parallel systems, where the scope for subtle error is greater.

The first goal of this work was to construct an operational model of a general purpose, parallel, object oriented system in order to ascertain the fundamental set of event classes that constitute its observable behaviour. Our model is built on the CSP process calculus and uses a subset of the Z notation to express some aspects of state. This alphabet was then used to construct a formalism designed to augment each object type description with the operational specification of an object's behaviour: Event Pattern Specifications (EPS). EPSs are a labeled list of acceptable object behaviours which form part of the definition of every type. The thesis includes a description of the design and implementation of EPSs as part of an exception handling mechanism for the parallel, object oriented language Solve. Using this implementation, we have established that the run-time checking of EPS specifications is feasible, albeit it with considerable overhead. Issues arising from this implementation are discussed and we describe the visualization of EPSs and their use in semantic browsing.

Acknowledgements

I extend my heartfelt thanks to my supervisor, Dr Russel Winder, whose strategic advice has guided me through the tricks and traps of academic apprenticeship and whose experience has helped to channel the exuberance of a tyro into this and other works. Without his help and confidence, much of this work would have been impossible.

Gratitude is due also, in no small measure, to (in alphabetical order): Peter Bates, Steve Cook, Siamak Masnavi, Oscar Nierstratz and Peter Wegner, who have read earlier drafts of this and related works and offered much useful criticism.

I would like to thank the Science and Engineering Research Council for providing the grant that made this research possible.

I dedicate this thesis to my parents, Alex and Priscilla Purchase, and my long suffering girlfriend, Diane Leung, for emotional and financial support in abundance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Problems

The tools and techniques used to detect and locate program 'bugs' have not evolved at a rate commensurate with that of programming language sophistication. Consequently, they are becoming increasingly inadequate. Furthermore, these tools have insufficient formality and rigour to meet the challenges posed by modern programming paradigms.

In computer science, a 'bug' is a defect in a program that causes it to deviate from the expected or desired behaviour. Bugs are a result of human error and, as such, are an inevitable consequence of human involvement in programming. They may be detected and located using a range of techniques: both pre-emptive, e.g. assertions, exception handling and testing; and curative, i.e. debugging. At run-time, the baneful results of (previously undiscovered) bugs can be mitigated by algorithm redundancy or exception handling, until the root cause can be corrected. All methods of detecting and locating bugs concentrate on establishing the existence and the nature of behavioural deviations. The techniques used to do this vary with the programming paradigm in use.

High level languages have developed enormously since the emergence of the first commercially viable examples, e.g. FORTRAN and COBOL, in the mid 1960s. These, somewhat monolithic, designs revealed a need for greater support of procedure based abstraction, leading to a multitude of languages supporting *procedure oriented* (*procedural*) programming, including: FORTRAN77, ALGOL, PL/1, PASCAL, and C. The procedure oriented (PO) paradigm views programs as collections of interacting functions and procedures. The nesting of procedural invocations is the principal structuring mechanism and design methodologies are based on functional decomposition. In PO programming the emphasis is on the *verbs* of the problem space, although the aforementioned

languages support data abstraction to some extent, data structures are passive and typically unencapsulated. The first language to fully support abstract data types (ADTs), SIMULA, began an evolution in the form of programming languages and design techniques, in which the emphasis passed from conceptual verbs to *nouns*. ADTs, encapsulated data structures with associated sets of procedures having exclusive access to this data, were originally defined as special case singletons, e.g. as in Modula2—in which each defined ADT represents one individual instance of the entity. The trend continued with *object based programming* (OBP), as supported by such languages as CLU and Ada, in which ADT *objects* are indistinguishable from the primitive data types offered by the language—i.e. they are first class data types. These ADTs can support multiple instances without a type manager, are fully encapsulated and separate the specifications of a type (its obligations to its clients) from its internal definition. It was not until the inception of Smalltalk, that true *object oriented programming* (OOP) began. Smalltalk combines the benefits of OBP with: inheritance, a hierarchical classification and composition mechanism that allows factorization of object commonality and object reuse; and invocation by dynamically bound message passing, that promotes full polymorphism and heterogeneous data structures. Although these individual features were not unique to Smalltalk at the time of its creation, they were combined in a unique way to form the object oriented computational model. Since then other object oriented languages have emerged, e.g. CLOS, Eiffel, C++, which have perpetuated the object oriented computational model through the provision of first class objects, inheritance and dynamically bound message passing [Weg90, PW91a].

This evolution of programming language design, or paradigm shift, has been characterized by increasing abstraction (and levels of abstraction), *locality*[1], potential for reuse, user imposed structure and reliability of programs. It has produced many new concepts which are changing the way in which programmers and designers work, and the types of mistake they make. Unfortunately, the functionality of debugging support systems like exception handling mechanisms and debugging tools are not keeping pace with this evolution. They have no new functionality to combat the new types of bugs from which object oriented applications are suffering [PW91a].

The failure of debugging tools to keep pace with developments in language design is not a new phenomenon. Over the last two decades, programming support tools like debuggers have seriously lagged developments in systems and language design [Mod79, MPW89], the vast majority of tools owing more to the assembly language debuggers of the mid 1960's than to any serious study of modern debugging requirements (e.g., [Mod79, BW83, BS73, vT74, Ves86, GD74]). Debugging tools seek to convey understanding of program behaviour—but they routinely fail to support activities shown to be central to the task of debugging (see Section 2.4): the conveyance of behaviour at

[1] The isolation of implementation details to avoid implicit dependencies between objects.

multiple levels of abstraction, defined by the paradigm, the language and the user; the comparison of actual and intended behaviour; and the formulation and testing of bug hypotheses. This failure has been a major shortcoming in the past; its continuation could be disastrous in the light of the new problems presented by parallel object oriented systems.

This problem is exacerbated by the introduction of parallel and distributed object oriented languages. The object oriented model is seen as a 'natural' means of harnessing parallelism, because of the analogy between processes and objects (and message passing and inter-process communication). Such languages and systems introduce the problems of asynchrony, non-determinism, latency of communication, deadlock and starvation [GKY89]. Furthermore, the introduction of parallelism may interact with the object oriented model [NP90] producing more problems. While a considerable body of research has considered the problems of debugging parallel systems, few have addressed the problems of debugging and exception handling in parallel object oriented systems.

Exception handling tools have also developed little since their inception with the seminal work of Goodenough [Goo75] in 1975. Although a great variety of mechanisms have flourished in various languages, most adhere strongly to the themes raised in Goodenough's work—many do not even implement this completely [PA90]. One genuine breakthrough was the coalescence of exception handling and state based assertion proposed by Meyer in the object oriented Eiffel language [Mey88, Mey89]. Despite the increasing use of such mechanisms in sequential object oriented languages [Don88], and recent cases of those for distributed systems [Lac91]—no research, to our knowledge, has established that state based assertion is the optimal means of detecting exceptions, or has examined alternative techniques. Furthermore, little work has been done to investigate the effects of parallelism on exception handling.

How can we detect behavioural deviations (bugs) in parallel object oriented systems and gather information to assist in their location, while taking into account the unique features possessed by these systems? How may we indicate to a debugger or exception handling system, with some rigour, what behaviours are considered correct? How can they check that these, and only these, behaviours occur? Is it feasible to annotate systems with these specifications to make permanent the association between specification and target object and enhance the users's understanding of objects? In short, how can we formalize the detection of bugs in parallel, object oriented systems?

1.2 Thesis

This work is motivated by a need to establish a method of improving the power and formality of bug detection and location techniques for parallel, object oriented systems. The crux of bug detection

and location is a knowledge of intended behaviour, actual behaviour and how they differ. A general purpose medium for specifying and describing behaviour in parallel object oriented systems is, therefore, critically important. A medium is required, with necessary and sufficient facilities to describe all possible behaviours and account for all of the special features of that paradigm. Furthermore, some means of comparing such specifications with actual behaviour and analyzing the extent to which they are satisfied is needed. The specification medium should be complete, unambiguous and equally applicable in exception handling and debugging contexts. In addition, the feasibility of the technique should be demonstrated though some form of implementation.

We assert that it is feasible to use unifying, operational specifications as a means of enhancing the power and formality of bug detection and location. Such a medium can be used to define the *observational* norm for a given application, in both debugging tools and exception handling mechanisms. The degree of adherence to this norm can then be used to locate bugs, or dictate any corrective action required to mitigate the error at run-time. Furthermore, we submit that the formal modeling of a parallel, object oriented system, needed to formulate such a medium, is a valuable exercise in itself.

1.3 Structure

In each of the chapters that follow, the points previously presented are substantiated with evidence and conclusions are drawn which establish the motivation and background for the following chapter. This document, however, does not necessarily reflect the chronological order in which the work was conducted, rather it indicates the underlying flow of reasoning.

In the survey (Chapter 2) we attempt to taxonomize two types of tools used to detect bugs: debugging tools and exception handling mechanisms. We illustrate: the variety of the available tools; the dilemmas facing their designers; the commonality between the tools; and their deficiencies—especially those which they share. Having established a need for greater rigour in both types of tools, we develop a formal, event-based, observational model of a parallel object oriented system in Chapter 3. This chapter helps to explain the unique features of such systems and to define and relate the terminology used throughout the work. This leads to the first application of the model—the basis of an operational formalism for the behavioural specification of parallel, object oriented systems. This formalism, event pattern specification (EPS), is detailed in Chapter 4.

The design of the EPS formalism is presented after a careful discussion of its design goals and applications. Two applications for EPS are considered: a means of behavioural hypothesis

generation in a debugging tool; and a means of supporting behavioural assertions in an exception handling mechanism. The design and implementation of the latter is presented in Chapters 5 and 6 respectively. In these chapters, particular emphasis is placed on the design of an exception handling system with minimal impact on the underlying language and the separation of pattern matching and constraint proving activity in the implementation. In Chapter 7, we consider related work in the fields of object oriented formal models, specification media and exception handling techniques. We conclude, in chapter 8, with a summary of our activities, findings and contributions. Furthermore, we analyze the limitations of our work and present some directions for further research.

Various typographical conventions are used in this document in an attempt to enhance its clarity. All conventional text is set in the Roman typeface. Single quotes, '', are used to delimit informal or unusual usage of a word or symbol, whereas double quotes, "", signify quotations from written or spoken works. Use of the **bold** typeface denotes the title of an item in a bullet list. The *slanted* typeface indicates the use of uncommon terminology, or a term coined by the author. In the latter case, the glossary appendix (Appendix A) holds a full definition of the term and its first usage is accompanied by a brief explanation. By comparison, the *italic* typeface depicts emphasis or mathematical entities. Finally, the `typescript` and sans-serif fonts are used to represent programming language keywords and variable types (for which an instance of the type must be substituted) respectively.

Chapter 2

Detecting and Locating Computer Program Errors

2.1 Introduction

2.1.1 Bugs: Cause and Eradication

As it pertains to computer science, a *bug* is a defect in a program's specification, design or implementation that causes the latter to deviate from the expected or desired behaviour when executed. This deviation causes the program to enter an erroneous state and, eventually, to exhibit this error externally—i.e. to *fail*. 'Bug' is a somewhat unsatisfactory term, lending a degree of autonomy and anthropomorphism to such defects which is inappropriate and suggesting evasion of responsibility on the part of the programmers [Thi85]. However, by virtue of its continued acceptance by the computer science community, we adopt it here. All bugs are manifestations of human error. Specifically, they are caused by the "human inability to perform and communicate with perfection" [Deu79]. Consequently, bugs in computer programs are an inevitable consequence of human involvement in programming [Kel88, vT74]. They arise because of human limitations, particularly in communication, perception, reasoning and memory [Mod79, CBM90, Sha82, Sen83]. The techniques used to combat bugs and to mitigate their consequences can be broadly classified into three types:

- **Prevention**, those which attempt to minimize the probability of introducing bugs, e.g. language design, formal verification, rigorous design procedures, automatic programming and the use of executable specification languages;

- **Pre-emption**, those which maximize the early detection of bugs in the executable, e.g. exceptions, assertions, and testing. The former may also allow recovery from bugs or other run-time anomalies, e.g. algorithm redundancy; and
- **Cure**, locating and correcting bugs, once their presence has been determined by pre-emption or through normal usage—debugging.

The justification for research into the latter two techniques, given the progress made concerning the first, is explained in this chapter. We present a survey of the available tools for imperative systems which support these latter techniques and discuss the problems they face. We consider especially the additional problems introduced by the usage of such techniques in parallel, object oriented environments.

2.2 Prevention

2.2.1 The Expensive Inevitability of Error

It is tempting to assert that, of the three techniques introduced in the previous section, prevention alone merits research because once a foolproof preventive technique is found, the need for others is obviated. Dijkstra [Dij72] uses this argument to claim "...good programmers should not waste their time debugging because they should not introduce bugs in the first place...". Similar views are asserted by Backhouse [Bac86]. There are three flaws with this argument. In practice, formal verification and design techniques are expensive [MH89] and the former is constantly outstripped by developments in programming language sophistication [CS89, MC88, MFS90, Sch71]. Secondly, such preventive techniques rely on the correctness of an initial specification and are themselves susceptible to human error. Consequently, they cannot guarantee absolute correctness [Som89, Yeh77]. As Van Tassel asserts: "a bug-free problem is an abstract, theoretical concept..." [vT74], especially with large, non-trivial programs [Sen83] and those which make extensive reuse of existing components. Finally, design is an *iterative* process, not a linear one—thus were it to be relied on completely, the overhead of formal techniques would be infeasibly high, greatly encumbering design. Clearly, given the inevitability of human error, errors should be *expected* in all non-trivial projects and to deny the usefulness of tools to pre-empt, detect and locate bugs caused by these errors is futile. Instead, actions should be taken to facilitate the early detection and correction of such defects to minimize their cost and increase the likelihood that they may be avoided in future [Kel88, vT74, BS73]. The expectation of bugs is the *raison d'etre* of all techniques based on pre-emption and cure.

The cost of bugs to programmers and their clients is vast. Commercial programmers spend in excess of half their time debugging software [BS73, vT74, Ves86, Ves89] and over 20% of their companies' budgets [LS80, YC79]. A substantial amount of software is unusable in its original form as a result of software bugs. As little as 2% of of the software commissioned by some sections of the U.S. government is fit for immediate use as intended [Cox86]. On occasion, even lives are lost as a result of software bugs [Neu91]. These figures are exceptional, but they are demonstrative of the fact that bugs and their removal are serious issues and not the sole preserve of a minority of bad programmers. A more detailed analysis of the type of bugs that occur and their cost can be found in [MPW89].

2.3 Pre-emption

It is widely believed that the earlier a bug is corrected in a program's life cycle the less expensive that bug will ultimately prove to be [Mye79]. Pre-emptive techniques are used with the assumption that bugs will always occur in non-trivial programming tasks. They have the goal of detecting and locating bugs at the earliest opportunity. One pre-emptive technique, exception handling, also provides a means of mitigating the effects of bugs and *anomalies*[1] detected at run-time. It is this technique that we examine in this section.

2.3.1 Exception Handling

An exception handling system is an infrastructure which supports a uniform and disciplined response to error, allowing one element of a program which has detected an erroneous condition to communicate with, and pass control to, another which is better qualified to resolve or minimize the damage caused by the problem. For example, designers of a module or object library are aware of the main limitations of their software: they know what can go wrong. In the event of failure however, it is often the client module that best knows what to do in response. Exception handling mechanisms are the 'glue' that bind these remote contexts, without compromising the encapsulation of either.

Typically, exception handling mechanisms support the distribution of *signallers*, by the user, at strategic points in the target program. Each signaller has an associated *assertion* which denotes a specification and a time at which the target must comply with this specification. Should these assertions prove to be upheld by the target at this time, the program runs as expected. However,

[1] Unexpected errors due to unforeseen behaviour of an agent beyond the control of the program, on which the program relies—for example hardware, or the user.

a violation of an assertion causes the signaller to signal an exceptional occurrence. Thereafter, the signal is propagated across invocation and process boundaries until an appropriate *handler* is found. The handler is a code segment designed to respond to the bug or anomaly which caused the exceptional occurrence.

It is important to realize that the purpose of exception handling systems is not to behaviourally 'patch' the program to compensate for bugs of which the programmer is aware at the time of deployment of the signallers. These bugs should always be rectified at the earliest opportunity. Rather, it is to protect program integrity from unknown bugs and anomalies and oversee the invocation of code segments, the applicability of which cannot be tested by any means other than invocation (e.g. any routines interfacing with users or other hardware).

Modern exception handling mechanisms still rely heavily on the protocols established by Goodenough [Goo75], despite the fact that some consider the work to be over exhaustive [PA90]. In this survey we stress the more practical and widely used facets of the work of Goodenough and others. Goodenough posited that like mathematical functions, many program operations or modules are *partial* with respect to their domain or range (or both) and should have some means of formally defining this partiality (assertions) and what happens if it is exceeded.

2.3.2 The Case for Exception Handling

Some feel that exception handling features compromise the formality, reliability, readability and simplicity of programming languages [Hoa81]. We feel the first charge is justified and consequently address it in this thesis. However, the last three charges are not a reflection of the inclusion of exception handling mechanisms *per se*, but of their poor implementation. Properly implemented exception handling regimes enhance program readability, reliability and simplicity. Assertions reflect the invariants of a system, allowing signallers to be designed as annotations which help convey the program's purpose and function—improving readability and ease of understanding. Signallers are *active* comments which help to document programs and, unlike traditional documentation, are (necessarily) always up to date [Mey89]. Thus, they help to satisfy a general need for better program documentation [Knu63] and allow "the purpose of the portion (of a program) [to] accompany the portion" [Sen83]. Handlers protect programs from failure, which may arise from it being used in unexpected ways, by insuring a 'sane' response to unusual circumstances [CW89, Goo75, Kel88]—thereby improving reliability. They may also be used to support software fault tolerance. Exception handling mechanisms can help to separate exceptional cases, those requiring extraordinary computation, from conventional code—improving simplicity. Furthermore, the effort required to

formulate assertions during program design can reveal the presence of bugs during both design and coding [Knu63] and give programmers a new perspective of the target [Lis87]. At run-time, the violation of assertions can pinpoint a bug more accurately than debugging after program failure has been noticed.

Should programming languages actively support exception handling? Most of the facilities offered by exception handling mechanisms could be achieved using keywords and concepts already part of most programming languages. However, syntactic support (providing it introduces a minimum of new concepts) makes programs more readable, and run-time behaviour easier to understand [PA90, YB85], by emphasizing the documenting effect of assertions and separating conventional code from the extraordinary. Furthermore, such mechanisms ensure a consistent style of exception handling by providing a standard, formal mechanism for recovery and termination [KS90]. Once an exception handling convention is enforced like this, the responsibility of each module is clearly defined and fewer assertion checks need be made, therefore improving efficiency [Mey89, DPW91].

2.3.3 Alternative Uses for Exception Handling

In this survey we consider only those uses of exception handling systems pertaining to the detection and handling of erroneous conditions. Alternative uses for such systems have been suggested. Goodenough posits that exceptions could also be used: as a means of module instrumentation; as a technique for denoting the need for special interpretation of a module's result; a method of escape from nested loops; and a means of iteration through a dynamic, 'container' data structure (such as a list or bag). Liskov and Guttag [LG86] believe that exception handling should be a general purpose control structure. They provide many examples of the usage of exception handling mechanisms as decision constructs and multi-way case statements which exaggerate the separation of the case action clauses.

However, the majority view is that control flow of exception handling mechanisms is complex enough to warrant its usage only *in extremis* [Mey89, KS90]. To use the technique commonly, as just another language construct, would greatly complicate the ease of understanding of program execution. Furthermore, resorting to exception handling for the reasons suggested by Goodenough, Liskov and Guttag is frequently unnecessary. The special interpretation of a result can be achieved by incorporating a flag into its data structure [PA90]. Escape from a series of nested loops by violation of conventional loop constructs is undesirable in any event and iteration constructs exist to facilitate the traversal of dynamic data structures (e.g. CLU's iterator function [LS79]). To conclude, exception handling functionality should be reserved for the signalling and handling of run-time anomalies and bugs [KS90, Mey89].

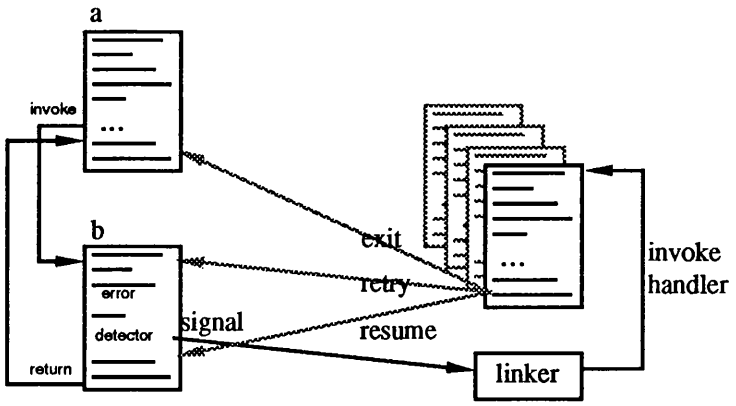

Figure 2.1: Phases of Exception Handling

2.3.4 A Framework for Exception Handling

The issues of exception handling are best considered by analyzing the phases of use of a typical mechanism. These are pictorially presented in Figure 2.1 and are summarized below. In this example, a program module *a* invokes another module *b* to perform some processing and the latter fails. The stages of exception handling and the issues they raise are:

1. **Placement.** The initial deployment of apparatus to detect (or *signal*) anomalous behaviour or conditions. What form should this apparatus take and how should signallers be deployed? How is the structure of exception handling mechanisms related to the broader issue of language design?

2. **Detection.** The means by which an exceptional condition is recognized and the responsibilities of the recognizing agent (i.e. signalling). How should anomalies be detected and signalled?

3. **Linkage.** The process of propagating a signal and finding the appropriate handler to deal with a signalled problem. How should handlers and signals be associated or *linked* flexibly? How can the usage of specific signal/handler pairs be associated with a given activation context, to allow context specific handling of certain exceptions?

4. **Handling.** Dealing with a signalled problem. How can a handler be passed information concerning the details of this particular anomaly? What strategy and control flow should the handler adopt—termination or resumption? Some control flow examples are shown as grey arrows in Figure 2.1.

In the following sections we consider these issues in more detail. We examine how they are addressed by existing systems and summarize the functionality of these systems in Table 2.1. In this table, as with all others in this chapter, the abbreviation *n/s* indicates that some information is not specified by the cited publications and *n/a* shows that a classification is not applicable to this example. A blank entry indicates that the system in question provides no functionality of the type specified. In Table 2.1, the first two entries have *n/a* in their host language columns because they represent general-purpose models and have no host language.

2.3.5 Placement and Language Design

The language design of exception handling (see the *Exception* heading of Table 2.1) has yet to be standardized and many diverse examples exist. In older models and languages (i.e. Goodenough's model and those from Poly [Mat85] and CLU [LS79]), signals are untyped names, or constants, which enumerate the potential exceptions. These are used in conjunction with `raise` statements, to broadcast news of the exception beyond the local block. The appropriate handler is chosen on the basis of the signal name, as it is made available on exit from a block. In more recent, object oriented examples, signals are data objects with type and scope [PA90], which are *thrown* from the detecting context to the handling context. These are termed *catch-throw* mechanisms and originate from LISP [DPW91]. This representation, coupled with that fact that exceptions have to be explicitly declared, makes the entire mechanism more amenable to static type and consistency checking. Indeed, the C++ [KS90] and Smalltalk-80 [Don90] systems use signal type (or class) to achieve most of the compile-time checking of exception handling. As Goodenough [Goo75] recommends, some [Don90] also use signal type as a way of communicating what control flow model the handler should follow (see Section 2.3.8). Although it is often advantageous to defer choice of the handler to the linkage mechanism (in an execution context potentially far removed from the seat of error and with greater scope for deciding how to respond), it is often safer if a raised signal can 'insist' on being handled by a sequence of instructions that will lead to its ultimate termination, as opposed to program resumption.

In Goodenough's original model, the parameterization of signals is not considered. However, as Table 2.1 attests, many modern systems have adopted parameterized signals as a means of providing a handler with information concerning the context of an exception. This helps to avoid global data structures (in which context information might otherwise be kept), increase modularity and enhance the flexibility of handlers. Signal parameterization is particularly expedient in object oriented systems. Therein, signal parameters are often rewritten as constructor arguments from

which to build a signal object with an internal state representing its context (see the column labeled *Scope* in Table 2.1).

Out of necessity, the semantics of some exception handling systems are rather convoluted, especially if handlers may themselves have exceptions raised within them (labeled *recursive* in Table 2.1). However, one of the goals of these systems is to simplify the handling of erroneous cases; this can be achieved by separating the seat of detection of a problem from where it is handled. A few of the surveyed systems aid (or even enforce) this separation (labeled *separate* in Table 2.1), by providing syntactic support for it. The optimal extent of this separation is a matter for debate (see [DPW91]). However, it is widely recognized that signallers and handlers must avoid detracting from, or obscuring, the main source code; otherwise maintenance will be hindered [PA90]. Complexity is the single worst enemy of software reliability [Mey89].

The unification of a language's software exception handling system, with a mechanism for handling other faults such as 'low-level' exceptions (failure of the host operating system, or hardware faults), was first suggested by Goodenough [Goo75]. Such unification is advantageous since the two mechanisms have similar goals and, once combined, should simplify the host language [Knu87]. Despite the advantages of this uniform approach (denoted *unified* in Table 2.1), it is absent in a surprising number of systems. This is partly because, in all but two cases, the surveyed systems are designed only to detect exceptions synchronously (labeled *sync* in Table 2.1). That is, a signaller's assertions are evaluated only when it is executed, so the signal can only be raised at a certain time during the execution of the host block, as determined by its location in that block. Asynchronous detection (labeled *async* in Table 2.1) adheres more strictly to the concept of invariants, in that signallers may raise a signal at any time during the execution of the host block when the assertion that they embody is violated. Asynchronous exceptions are generally more powerful. Although, on occasions, blocks may temporarily violate invariants (legitimately) during intermediate stages of execution and thus the temporal scope of signallers must be subject to user restriction. In Eiffel [Mey88], this is achieved by giving each type of signaller a fixed (non-universal) coverage. All 'low-level' errors occur asynchronously with respect to the program and thus it is difficult to unify them with entirely synchronous detection schemes. One unification technique involves establishing a set of predefined software signals (labeled *predef* in Table 2.1) which represent all possible low-level exceptions; this is possible since the range of such exceptions is usually small (e.g. the exception set of the Motorola MC68000 microprocessor numbers only 47 [Wil85]). At run-time, any low level exception that occurs is mapped on to the appropriate, predefined high level exception and signalled asynchronously. Predefined signals may also provide an environment with *generic* exceptions—i.e. those that merely indicate the existence of an exceptional condition, or some broad class thereof, without providing any details—thus helping to enforce encapsulation

(see Section 2.3.8).

2.3.6 Detection

All currently available signallers use state based assertions, i.e. they detect anomalous conditions by evaluation of expressions in the host language, which check certain aspects of state. Little study has been done to determine if this is the optimal form for assertions. It is certainly an imposing limitation to restrict the expression of specifications—for that is what assertions are—to conditions of system state, defined in the host language.

The vast majority of signalling mechanisms are *explicit* (labeled *explicit* in Table 2.1). If an assertion is violated, a signal is explicitly raised. For example, a signaller statement might resemble:

```
if (something_wrong(x)) then raise problem
```

Alternatively, in some object oriented systems, a signal object may be explicitly created and sent the message *raise*.

Only the Eiffel system [Mey88] permits *implicit* exceptions, i.e. a signaller which automatically raises an exception if its assertion is violated, without the need for a `raise` command. For example:

```
invariant <Signal Name>:  not(something_wrong(x))
```

This syntax variation may seem slight, but it does enforce a consistent mapping between assertion and signal, it is a greater aid to program documentation and it is more disciplined. In contrast, the `raise` statement is more likely to be used inconsistently, or for the wrong reasons [Mey89].

Eiffel is also unusual in directly supporting, via its *contract* metaphor, Goodenough's domain and range checking protocol. It does this through its `require` and `ensure` constructs. Furthermore, Eiffel's asynchronous `invariant` construct directly supports the partiality of data representations cited by Goodenough and Philbrow and Atkinson [PA90]. It is illustrative of the way in which Eiffel uses signallers as active documentation.

Host	*Exceptions*		*Linkage*		*Handler*		*Ctrl*
Language	*Signal Definition*	*Mechanism*	*To Handler*	*Extent*	*Models*	*Scope*	*Flow*
n/a [Goo75]	typed, no params predef	explicit recursive sync, unified	dynamic by name 1:1	block	exit, terminate, resume, explicit propag	signal params	2-way
n/a [Knu87]	names typed params	explicit sync separate	static by name 1:1	block	smooth & immed termination	signal params	2-way
CLU [LS79] [LG86]	names typed params predef	explicit recursive, separate	static, by name catchall, 1:1	expr	resume, local terminate implicit propag	signal params	1-way
Poly [Mat85]	unique type string param predef	explicit implicit sync	static by name 1:1	block	exit, implicit propag is termination	global	1-way
SML [Wik87]	declared names, typed params, predef	explicit separate sync, unified	dynamic by name 1:1	function	termination implicit propag as generic	signal params	1-way
Mesa [MMS78]	declared, typed, typed params predef	explicit sync unified	dynamic catchall by name, n:1	expr	resume, retry, implicit propag, exit	signal params	1-way
Ada [FM89]	declared, names, no params predef	explicit recursive sync	static by name 1:1	block	termination implicit propag as generic	global	1-way
PS-Algol [PA90]	declared, typed typed params predef	explicit sync, async unified	dynamic by name catchall, 1:1	expr	retry, termination, explicit propag	signal params	1-way
C++ [KS90] [Koe90]	declared object typed params	explicit sync	dynamic by type catchall, 1:1	block	exit implicit propag	signal object state	1-way
Eiffel [Mey88] [Mey89]	untyped names no params predef	implicit unified sync, async	dynamic by location 1:1	method, class	termination retry, resume implicit propag as generic	host object state	1-way
Smalltalk [Don90]	subclasses of exception class predef	explicit unified sync	dynamic by method 1:1	expr, class	exit, retry, resume, explicit propag	signal object state	1-way
O/W Smalltalk [DPW91]	signal object, exception object param	explicit unified sync	dynamic catchall, n:1 by name	block	explicit propag, terminate resume, debug guided.	signal object state	1-way
Beta [MMP89a] [DPW91]	named handler method, no params	explicit sync recursive	static guided by method 1:1	program, class, object, method	resume, retry, debug, smooth termin -ation, debug, guided.	host object state	2-way

Table 2.1: Characteristics of Language Based Exception Handling Systems

2.3.7 Linkage

Linkage concerns the mechanisms used to bind a raised signal with an appropriate handler. This includes the association of handlers with signals and the means by which exception handling constructs impose localized signal to handler mappings. Many early exception handling systems favour a dynamic association scheme between detected signals and handlers [Goo75], e.g. the computed linkage facility of PL/1 [PA90], or those based on searching the invocation stack for handlers at run-time as in CLU [LS79] or Eiffel [Mey88]. Of late, however, the enhanced accountability of *static linkage*, i.e. performing the signal-handler binding at compile time rather than during execution, has won favour [Knu87] and has been adopted by a few of the surveyed systems.

Syntactically, linkage is typically achieved by providing a namelist of expected signals (or signal types) and their associated handlers. This *mapping* is then associated with the code segment over which this linkage is required. This code segment is known as the *extent* of the signal/handler mapping. The flexibility and granularity with which the extent can be defined varies with the architecture of the underlying language and the signal propagation scheme in use. Usually, it is related to the unit of modularity of the host programming language, hereafter abbreviated to *module*. Most systems allow mappings to be associated with blocks. Other more advanced ones allow mappings to be associated with individual expressions, although this is difficult to achieve unobtrusively [Goo75] or without violating separation.

A handler search (done at run-time or compile-time) is most often guided by signal name alone (labeled *by name* in Table 2.1). Some searches, like the catch-throw mechanism of C++ (see Section 2.3.5), are type driven and others, like Eiffel, are resolved purely by syntactic location of the handler.

One practical obligation of a linkage system is to provide default exception handlers in case signals are raised and never 'claimed'. In some static linkage systems, raising a signal that is not explicitly handled somewhere is illegal and results in a compile time error. Consequently, the need for default handlers, either language or user defined, is reduced. However, most linkage systems have default handlers for unclaimed signals and some allow the user to define 'catchall' handlers (labeled *catchall* in Table 2.1). This construct is useful for handling classes of signals which would be too numerous to handle individually, e.g. in unified systems where many predefined (low-level) exceptions may occur, but all require the same response. A few systems support this functionality directly by permitting multiple signals to be mapped on to the same handler (labeled *n:1* in Table 2.1).

2.3.8 Handler Definition

Handlers differ in three important respects:

- **(Lexical) Scope**: how they obtain information about the details of the exception (covered in Section 2.3.5, see column *Scope* of Table 2.1);
- **Medium**: how they are defined; and
- **Model**: the control flow they impose on the host program, after handling is accomplished.

Handlers acquire details about the exception to which they must respond in a variety of ways. Some mechanisms amass contextual information in a series of globally accessible variables when a signal is raised (e.g. Ada, Poly; labeled *global* in Table 2.1), some make the state of the failing module accessible to the handler as if it were being executed in-line (labeled *host state* in Table 2.1) and others use signal parameterization (see Section 2.3.5). In those systems which represent signals as objects, it is most convenient to make contextual information a component of the signal object. These latter techniques are generally better, because they avoid violation of encapsulation and they allow the contextual information to be more dependent on the signal.

Most exception handling systems define handlers as conventional modules, i.e. blocks or sequences of executable code. This increases the orthogonality of the system and avoids the need to add new language concepts just for exception handling. Little study has been done to determine if this is the optimal format for handlers. If handlers are conventional modules, should they be able to signal exceptions, i.e. should exception handling mechanisms be recursive? Some are (labeled *recursive* in column *Mechanism* of Table 2.1), but this facility greatly complicates control flow. Goodenough's original proposal is itself unorthogonal in this regard: certain types of handler may raise signals, but some may not.

Six control flow models are commonly used in exception handling mechanisms (see the column headed *Model* in Table 2.1); these are listed below.

1. **Terminate**. The faulty context is abandoned and preparations are made to report the error, by whatever means are appropriate, before the module and program are aborted completely. Many exception handling systems default to this model as it is undoubtedly the safest. It is especially appropriate for range errors.

2. **Resume.** The context is repaired and control is returned to a point just after detection of the error. This return point is usually the statement just after the failed expression, or the statement following the *extent* of the handler (see Section 2.3.7). Many consider this model unsafe [KS90], as it can—and does—lead to further exceptions. Others consider it unworthy of direct syntactical support, as it can be simulated by conventional language constructs [PA90]. If the module's purpose was to return a result, resume may provide its own result in lieu of the one now lost.

3. **Retry.** Any damage done by the execution of the faulty module is repaired, or reversed by rollback (of a checkpoint). Changes are then made to the context to prevent recurrence of the problem. Finally, the faulty module (the code within the handlers extent) is re-executed, in its entirety, from the beginning. This model is especially useful for domain failures, particularly if the scope of the handler includes the arguments of the faulty module.

4. **Delegate.** As retry, but the re-executed module is not the original, but one specified by the handler [YB85]. This model directly supports software redundancy.

5. **Propagate.** The local context is abandoned and the exception is propagated to the enclosing block or module. This reflects a natural and powerful need to deal with an exception first locally, and then globally [Goo75]. Many systems support *explicit propagation*[2] (see Table 2.1), but the somewhat less safe *implicit* (or *blind*) propagation, in which all unclaimed signals are automatically propagated without alteration, is also used. The latter can be highly disadvantageous, as signals bearing information about the failing of an operation can be inadvertently propagated to a client context which should be unaware of the internal workings of a server, thus violating encapsulation. Some forms of implicit propagation avoid this problem by implicitly propagating only *generic* signals (see Section 2.3.6). Because these signals impart no information about the cause of the problem, they may be freely distributed without risk of violating encapsulation.

6. **Debug.** Suspend the faulty context and spawn a debugger to examine the context of failure. Clearly this is only a development option, but some systems support it.

How a given handler establishes which of these models to use, using only conventional module syntax, is an important consideration. Most systems are obliged to allow syntactical caveats to circumnavigate this problem.

[2] In a system supporting explicit propagation all signal propagation must be done manually, giving the user the opportunity to propagate a new signal at a higher level of abstraction than that received by a failing server.

The model used depends entirely on the handler ultimately executed in response to a signal. However, some systems allow a signal to constrain the models that may be used to handle it (labeled *guided* in Table 2.1). For example, in ObjectWorks Smalltalk [DPW91] (see also Section 7.4.1), instances of the class *Signal* have a boolean instance variable *mayProceed* which indicates whether or not they may be resumed.

Whatever model is used, exception mechanisms attempt to make the control flow explicit. Control flow discipline is of the utmost importance, as is its understandability. Goodenough's model fails a little in this regard since, although it lends itself to compiler checking, some of its mechanisms are analogous to non-local 'goto' constructs and are consequently ill disciplined. Goodenough's mechanism is also very source code obtrusive because it demands a high level of explicit signal propagation. This demand is itself compromised by the **pass** construct, which overrides explicit propagation, although it does force a review of involved modules when a new exception is added to the system.

To be effective, models involving partial termination must guarantee consistency of the host environment. This can be difficult if the exception occurs asynchronously. One approach called *smooth termination* is described by Knudson [Knu87]. It allows each terminating block to cleanup after itself automatically. Knudson argues that for smooth termination of all nested, parent blocks to an exception context, upward propagation is not enough in static exception systems. Upper (lexically outermost) blocks should be allowed to initialize first, passing control to lower blocks, which propagate back to the higher blocks. He describes the concept of *prefix sequels* to overcome this by giving each exception handler an opportunity to cleanup before it is terminated. Knudson's system is not the only system to feature bidirectional control flow, but few others do (see the *Ctrl Flow* heading in Table 2.1) because of the immense complexities involved and the difficulty in understanding such systems.

2.3.9 General Problems with Exception Handling Mechanisms

Despite the wealth of features and benefits offered by the surveyed systems, many individual implementations seem to justify one or more of Hoare's criticisms (see Section 2.3.2). In particular, formality, which *none* of the surveyed systems address. We believe that the main problems with existing exception handling systems are as follows:

- **Weakness of Expression.** All systems use the host language to express assertions. Although this greatly eases the use of these mechanisms, it compromises their power and

formality, limiting anomaly detection to that which can be deduced by boolean, state expressions within the scope of the current module. Assertions are *specifications* of desired facts, and as such, may not be best expressed in a programming language. Other media offering better powers of abstraction and formality might overcome this weakness.

- **Poor Separation**. Many exception handling systems, notably that of CLU [LS79], make little distinction between signallers, handlers and primary module code, allowing all three to be mixed at will. This counters several of the foundations of exception handling mechanisms and obscures module semantics.

- **Lack of Discipline**. A few exception handling systems override the existing control structures of the host language in a somewhat unsafe manner. They are open to abuse from persons wishing to use the mechanism as an easy escape from heavily nested constructs, especially if signalling is explicit. They leave too much to the discretion of the user, leaving them open to ill conceived and inconsistent use. The 'raise' mechanism of Ada typifies this [Mey89].

- **Non-unified Error Handling**. To reduce complexity, assertion violations should be signalled and handled using an identical mechanism irrespective of where they originate. Retrofitted exception handling mechanisms (e.g. that of C++ [KS90]) often ignore or are unable to comply with this point, resulting in different mechanisms of hardware, operating system and software exceptions.

2.3.10 Problems Introduced by Parallelism

Few have yet considered the impact of increasingly popular concurrent architectures, and host languages, on the characteristics of exception handling mechanisms. Several of the assumptions that can be made with sequential systems—the concepts of determinism, synchrony, and locality—no longer apply, exaggerating some of the above problems and creating new ones, such as those listed below.

- **Non Determinism.** How can we maintain rigourous state based assertions, with some support for expressing limited non-determinism, in the host language? How can assertions express constraints on the available parallelism?

- **Asynchrony.** Should one process asynchronously spawn from another and then detect an exceptional circumstance, that it cannot handle, how should it propagate the signal? Should it be forced to synchronize with its parent to deliver the signal? What if the parent has

subsequently completed or been terminated [DPW91] ? How can the consistency of a system in which multiple exceptions are concurrently signalled be maintained?

- **Distribution.** How are signals to be propagated over process, processor and even site boundaries? If a number of threads are waiting in a monitor and the holder of the lock signals a fatal exception, how should the dependent threads be treated [DPW91] ?

These problems and others, increase the degree of complexity required in an implementation of a parallel language with exception handling, making it more difficult to ensure that such a mechanism does not make programs more difficult to understand. Ironically, since the scope for subtle errors in parallel systems is greater than that in sequential systems, one could argue the need for exception handling systems is increased.

In some ways, parallel systems would seem better candidates for exception handling systems as they offer natural abstractions for software redundancy. Furthermore, they might allow handlers to be executed in parallel with the exception halted processes [BGH+89]. However, the probe effect [Gai85] might prove to be a problem by radically altering the temporal behaviour of conventional and exceptional cases.

2.3.11 Problems Introduced by Object Orientation

The high level of modularity introduced by object oriented architectures enhances and eases the use of exception handling mechanisms in many ways. It provides (apparently) natural constructs—the class and the method—about which to accumulate standard handlers and domain and range based signallers. Furthermore, as these are basically class annotations, they can be separated from the principal class methods without being lost. Inheritance offers a powerful means of factoring out commonality in handlers and ensuring an orthogonal response to error. Rigourous object models, like that of Eiffel [Mey88, Mey89], may enhance the formality of exception handling. Additionally, objects form the ideal representation of exceptions. They have an internal representation which can include details of the context of the signal and any requests or constraints about how it should be handled [Don90]. They may be assembled into inheritance hierarchies to represent the classification of exceptional cases and support greater understanding and reuse [KS90].

However, object orientation introduces some problems into exception handling. Object orientation itself reflects a higher level of abstraction, in programming, than the traditional procedural approach [Weg90]—state-based assertions do not reflect this enhancement. Ideally, we require some specification technique which provides a level of abstraction commensurate with the abstract

data types (ADTs) provided through object oriented programming. Indeed, an assertion mechanism which could formally express desired program behaviour at multiple levels of abstraction is needed.

Some object oriented languages have exception handling *retro-implemented*[3] and this introduces grave problems. For example, the C++ exception handling mechanism is weakened by its retro-implementation which: limits its choice of keywords; requires that it support antiquated linkers; prevents unification of the mechanism with (hardware and operating system) signals and prevents it from supporting the resumption model.

Systems which perform manual memory management also introduce problems. These basically concern the occurrence of exceptions during the execution of constructors. How can one recover a system which may consist of some partially constructed objects? Similar problems arise with destructors, where memory deallocation of partially allocated structures might reduce the integrity of the environment.

2.4 Cure

2.4.1 Debugging: Art or Science?

All programs are potentially bugged. The process of locating bugs and amending programs in order to eliminate them, in order that the program behave as intended, is termed 'debugging'. There is considerable disagreement in the literature regarding the nature of debugging skill. Analyses, such as those given by Model [Mod79], indicate that debugging is, or should be, based on a strict scientific method [BW83, Moh88, BFM+83] and many such methods have been proposed [BFM+83, BS73, Sch71, vT74]. Typically these are based on a cycle of behavioural observation, hypothesis and experimentation, until an amendment is determined which, when applied to the program, causes it to no longer exhibit the errant trait. However, there is evidence to suggest that, in reality, debugging is not this rigourous. The experimental evidence of Vessey [Ves86, Ves89] indicates that while novices use a context-free, model based approach to debugging, for most the process is dependent largely on experience and intuition, not on method. Indeed, Vessey establishes that persons with debugging expertise lack any conscious method—lending to the activity some semblance of an art form. Furthermore, Beizer asserts: "...Debugging demands intuitive leaps, conjectures, experimentation, intelligence and freedom." [Bei84].

[3] That is, being added as an afterthought, after the core language design was completed.

There is little disagreement that the primary, and most difficult, goal of all forms of debugging is *understanding* [Gai85, Mul83, Sen83]. Specifically, to understand what is happening during program execution, whether it is correct or not and, if not, how and why the program performed the erroneous actions. Debugging seeks to "attain understanding of causes of an error, or at least where behaviour of implementation varies from that desired" [BW83, Moh88, Bal69]. Such understanding requires that the user can predict the intended effects of program execution and can establish a relationship between the individual effects of that execution and parts of the program—this is essential in deducing the point of origin of a bug. To facilitate understanding, selected facets of program structure, behaviour and state have to be rendered more tractable to examination and manipulation than is normally the case. Only when this understanding is complete can the secondary debugging goals of proposing a solution, program amendment, and confirmation of that amendment, be achieved.

2.4.2 Debugging and Testing

Debugging shares many of the attributes of software testing [ST83], particularly the comparison of actual program interaction with that expected. However, there is a distinct difference between the two which is often misunderstood [Mul83] (e.g. as in [Bac86, Mod79]). As indicated by Beizer and others [Bei84, Mye79, Bru85], debugging and testing differ in goal, methods and psychology. Testing is the unsolicited attempt to verify program behaviour and to determine the presence of bugs in a planned systematic way; debugging concerns the localization and eradication of bugs once they have been detected (possibly by testing). Debugging requires a thorough knowledge of program specification, design and implementation; some forms of testing (black-box) only require details of the program's specification. The debugging process itself might involve making many tests, but such tests are secondary to the task of bug localization and are not conducted for their own sakes. Consequently, they cannot be considered as testing *per se* [Mye79, vT74, Deu79, Som89, GH88, Ham88, Las89]. We regard software testing as a separate issue and do not address it further in this thesis.

2.4.3 Methods and Means

In the measured, meticulous world of computer science, it seems odd that a problem facing computer scientists themselves should face such little examination. Yet, human debugging activity is an issue which few have theoretically addressed and even fewer have examined methodically. Of these, some use experimental evidence to determine the requirements of the debugging process

[Ves86, Ves89, Bru85, FM89], but others extrapolate from their own experience to yield such information [Joh83, ST83, Bat83]. These works are surprisingly scarce and ill referenced by the copious articles which purport to have implemented both original and useful debugging technology.

What experimental evidence there is suggests that debugging is a very personal activity: there is much variation in the debugging method used by different persons. Using verbal protocols to establish the problem solving behaviour patterns adopted by programmers debugging programs, Vessey [Ves86, Ves89] discovered that over 60% of debugging time is spent mentally executing the errant program and studying its output. The rest is spent planning (setting, canceling and achieving goals), knowledge building (gathering data about the program) and locating bugs (hypothesis generation, confirmation and code correction). Debugging experts, those who are effective (find bugs quickly and without error) and efficient (rarely have to start afresh, change debugging activity or change the area of program which they are investigating), mentally process code at a more abstract level than source statements and are better able to *chunk*[4] or cluster such information. They spend more time initially comparing intended and actual program output, studying source code and evaluating this information to form a mental model of the correct function of a program. However, novices leap into conjecture and hypothesis immediately. Once they form hypotheses, experts are more willing to discard them should they prove flawed—often however, their first hypothesis is correct. Vessey also establishes, using text comprehension theory on programs, that the further the bug is down the hierarchical structure of the program, the harder it is to detect and the longer it takes to correct. The serial location of the bug has no effect, however, because programmers do not examine programs serially—instead, they go to the module they believe is bugged with a top-down, breath-first search. Ultimately, the effectiveness of debugging depends on the abstraction gap between programmers' internal knowledge structures and the information available from the debugging environment.

In his experiments, Gould defines three types of non-syntactic (he asserts that syntactic bugs are trivial) bugs that commonly occur in FORTRAN programs and attempts to determine which are most difficult to locate and what debugging strategies are used [GD74]. He hypothesizes that the difficulty of debugging lies with the wealth and variety of information available—all of which can influence a programmer's strategy. He found that a strategy is chosen immediately, on beginning the debugging task. Often, programmers look for clichéd violations of language semantics initially, and then alter this strategy as clues are found. Mental models of program function are very important, and familiarity with a program speeds up bug location by a factor of three and reduces the error rate (counterintuitively, the two are associated). Once a bug is located,

[4] Chunking is the cognitive process of associating a series of objects into one object at a higher level of abstraction to save short term memory [Mod79], as one might abstract the concurrent letters 'c', 'a' and 't' into the single word 'cat'.

programmers are often confident that they know how to repair it. Gould concludes that, in many cases, debugging performance could be improved if programmers checked the differences between actual and expected program output more thoroughly. Consequently, the success of a debugging environment is dependent on the extent to which it supports this activity.

2.4.4 The Case for Debugging Tools

Traditionally, debugging is seen as a private activity which is often tackled, as the above findings indicate, with a set of personal techniques which have been painstakingly acquired by experience. Until recently, debugging tools were rarely used. All debugging was achieved by manual code amendment: instrumentation of user programs with 'print' statements to facilitate a greater understanding of execution progress [vT74, BS73, Sch71, Bat89]. This form of debugging is still common [AG89, Men87] and tools like *ctrace* [Kel88] exist to automate it. It is undeniably one of the most flexible debugging strategies.

Automated or not, the code amendment technique has serious limitations. Primarily, it is rather a primitive, ad-hoc method which promotes little pre-meditation or planning. Often it leads to guess work, time consuming edit-recompile-test loops and, if the amendment is flawed, further error [vV89, Mye79]. Worse, code amendment can *hide* timing bugs due to the *probe effect*[5] [Gai85]. The amendment process requires that the user is aware, in advance, of what parts of a program he wishes to gather information about—often this is not the case. Lastly, it is incumbent on the user to remove all amendments once the bug is discovered; if by some oversight this is not done, the amendments may cause failures of their own. Generally, these problems are exacerbated when one is faced with the more demanding job of debugging object oriented, or concurrent programs [CBM90].

Debuggers have none of the inherent problems of amendment. This is chiefly because most debuggers achieve visualization by temporarily altering the run-time image of a program. Furthermore, such tools have many additional uses: they may be used as a teaching aid to demonstrate programming language semantics [Chu83]; programmers can use them to demonstrate or explain the behaviour of an algorithm ("an explanation facility for a system is a integral part in ensuring that the system is understood and used correctly" [KG88]); they can be used in programming environments to assess the reuse potential of a software module; and during program maintenance to test program alterations.

Debuggers have received poor usage in the past because their importance has been considered as secondary to that of compiler, environment and OS development [MPW89, Moh88, BFM+83,

[5] The probe effect is the alteration in a program's time characteristics due to internal instrumentation.

Men87, Bal69]. Consequently, they were usually poorly documented, late in arrival and once available, partially obsolete because of their dependence on rapidly changing hardware or OS kernels (see Section 2.4.13). In addition, many debugging tools for high level languages have borrowed heavily, and inappropriately, from the techniques used in machine code debuggers, instead of deriving new methods for high level debugging. Furthermore, being targeted at systems programmers, their user interface was usually of such a poor standard as to intimidate novices [ST83]. Thankfully, there is some evidence that these trends are reversing [BEH88, Fel89, DP89]. Some believe that no matter how good the available debugging tools are, human nature will, to some degree, preclude their use [vT74]. Programming skill is frequently a matter of great pride for those that exercise it—to use a debugger might be construed by some as evidence of 'failure'. Others will regard the investment of learning to use a debugging tool as too high and return to ad-hoc techniques—to their cost [Bru85].

2.4.5 Design Requirements of Debugging Tools

Clearly, debugging tools should aim to enable more users to emulate debugging experts. This can be achieved by the provision of functionality to support:

- **mental execution**, through use of static analysis on source code and run-time behaviour monitoring to outline program structure and emphasize user abstractions;
- **experimentation**, by allowing the user to control (with *repeatibility*[6]) program execution[7] and to manipulate its environment, time frame, state and behaviour *on the fly*;
- **comparison of real and intended behaviour**, though the provision of a means of specifying behaviour and the ability to automatically compare actual behaviour with such specifications [Bat89];
- **hypothesis generation and confirmation**, by providing a means of expressing such hypothesis and an automatic way of testing them;
- **relating individual behaviours to parts of program source**, through co-visualization of program source, state and behaviour;
- **selective relaying of information**, by supporting the chunking and filtering of all information types, such that its volume is reduced, but the semantic content enriched [Bal69]; and

[6] Such that identical conditions, imposed by the user, yield identical results on each such execution.
[7] With no penalty for such control which might itself have side effects on program execution.

- **the individuality of users**, by providing ample scope for customization, adaptability and flexibility.

It should be remembered that, since usage of debugging tools is driven by adversity and often a degree of urgency, such usage should not exacerbate the situation. Debugging tools should be easy to use and of immediate help, otherwise users will not persist in their use [FM89, ST83].

2.4.6 Automatic and Manual Debugging Tools

Clearly, the optimum debugger design is one that provides the best support possible for the activities outlined above. Two different approaches have emerged: the manual and the automatic debugger. Automatic debuggers are systems that detect and locate errors with little help from the user, except to correct the error once found. Typically, they are knowledge based systems which either: use schematic knowledge of the host language to hunt for cliché (commonly occurring) bugs [Wer82, Har83, Sha82]; or make use of user provided, program annotations which describe the program semantics, in order to detect application specific bugs [JS85, Ada80]. Manual debuggers, by far the more common, merely serve to maximize the productivity of programmers in those activities described above. Our work only considers the latter in any depth. We feel that automatic debugging has severe limitations. Cliché systems are only of use in educational environments, where they may be applied to the programs of novice programmers. Frequently occurring cliché errors (for example the confusion of the equality and assignment operators, = and ==, in C) are often a sign of poor language design and should be dealt with by language modification. Annotative automatic debuggers suffer many of the drawbacks of prevention mechanisms and, in addition, can be very obtrusive. Currently, the best debugging agent is still the programmer [Mul83] and the most promising and widespread progress is from manual debugging techniques [CBM90].

2.4.7 The Universal Debugger

To examine the functionality of debuggers at a greater depth and to structure the discussion that follows on currently available debugging tools, we describe here an abstract model of a manual debugging tool and the concepts that underlie it. Other models of debugging tools exist, however none has yet considered their internal structure or been used as a medium for conducting a survey [CBM90, Bru85].

A debugging system involves five agents, all of which have set means of communication. These are: the debugging tool D, the operating environment ENV (for example the host operating

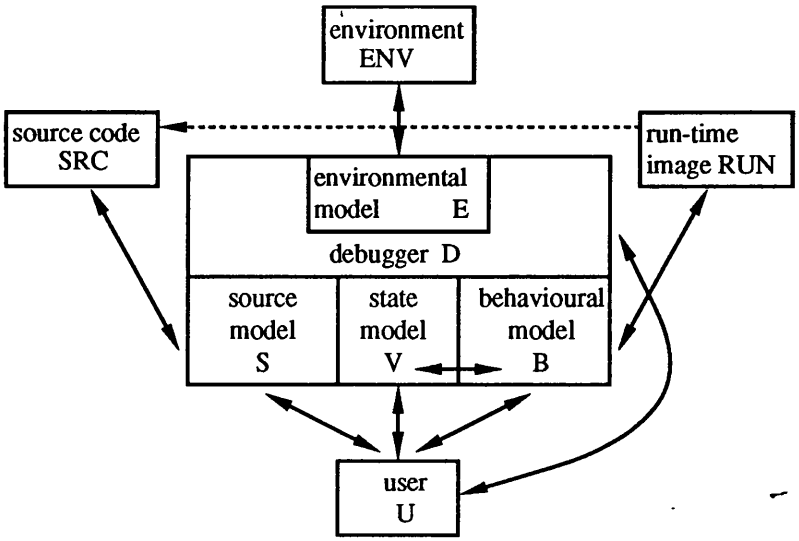

Figure 2.2: An Abstract Model of a Manual Debugging Tool

system), the source representation of the program being debugged SRC, the run-time manifestation of the program being debugged RUN (i.e. its process or thread) and most importantly, the user U. These are depicted in Figure 2.2

In order to help the user understand and manipulate all aspects of his program, the debugger internally maintains three models of it, which are partially visible to the user [BTM89]. The source model, S, is a symbolic representation of the program source code structure; V, the state vector model, represents the data objects of the program and B, the behavioural model, represents the programs run-time activity. Another model, E, is one the debugger maintains of the execution support environment, which is rarely accessible to the user. Connecting the agents are channels (solid arrows in Figure 2.2) along which information flows bidirectionally; each channel is named after the two agents it links. The dashed arrows of Figure 2.2 represent data dependencies.

In the sections that follow we consider each of these models and as its associated channels in turn, illustrating each feature with practical examples and, where possible, contrasting different approaches. The channel set involving the user (channels US, UV and UB) is dealt with in Section 2.4.12 which directly addresses user interface issues. Tables (with references where there is sufficient room) will depict how the surveyed tools fulfill the requirements identified by these models.

Debugger	*Source Model*	*Navigation*	*Manipulation*	*Ref*
Dbxtool	source text	line#, procedure, object search		[AM86]
Cbug	source text	procedure		[Gai85]
Jdb	source text	procedure		[WN88]
Pi	source text	line#, procedure		[Car86b]
Exdams	source text	line#, procedure, object search, cross ref, flowback	edit, recompile	[Bal69]
IDE	source text		edit	[Chu83]
Track	source text, inheritance chart	method, object search cross reference	edit, recompile	[BH90b] [BH90c]
Ups	source text		edit	[Bov86]
Gdb	source text	line#, function		[Sta88]
Object/Action	source text	class, method		[LL89]
IC*	source text schematic outline	data flow, invariant analogical graph	check consistency	[CC89]
Amoeba	source text	procedure, data info		[Els89]
O_2	source text	line#, method object search, data info	edit	[DP89]
PROVIDE	source text function outline	procedure	edit, recompile	[Moh88]

Table 2.2: Source Model Support in Modern Debuggers

2.4.8 The Source Model

The source model is an abstraction of the target source code maintained by the debugger, through which programmers may obtain better structural understanding of and manipulate textual aspects of their program. Typically, the model aids user source *navigation* using a variety of abstractions, from line numbers to data flow dependencies (as in the flowback system of Exdams [Bal69]) and it can facilitate amendment of a program, once any errors are located. The model is usually supported by static analysis and through it, some debuggers offer functionality such as: cross referencing of program objects (i.e. procedures, data items), by usage or inter-dependencies; object search; inheritance graphs (in object oriented systems); and syntactic outlining. Manipulation through this model includes facilities such as source editing and automatic recompilation. Table 2.2 shows which of the debuggers surveyed possessed any of this functionality[8].

Although it is argued that a key aspect of debugging is understanding a program, few debuggers provide any functionality specifically supporting the source model other than a conventional textual listing (these are not listed on Table 2.2). This is, in part, due to the fact that source understanding and manipulation is traditionally supported in software development environments, by tools other than debuggers, e.g. the Smalltalk-80 browsing environment [Gol83, KP86, WP88],

[8]Readers may note that the survey is not restricted to the debuggers listed in Table 2.2, one may infer that the debuggers not listed here do not support the source model in any way.

R↑N [CCH+87], the Cornell Program Synthesizer [TR81], PIGS [PN81], GraphTrace [KG88] and automatic flowchart generators [Knu63]. Despite this, this omission is a failure of the debugger to fully support the debugging task, especially if the semantic review of programs is not supported by any other tool available to the user. Debuggers also fail to aid understanding by omitting functionality that could be implemented by static analysis, to perform such tasks as the extraction of variable 'slices' [Wei82, Gra83] and consistency checks. One interesting exception is the debugger for the IC* system [CC89], an unorthodox language based on multiple threads of non-deterministic forward chaining invariants. Static analysis is used in this tool to produce hierarchical, graphical schematics of user programs and to offer extensive navigation facilities. If required, these representations may be adorned with data flow annotations and the consistency of the program checked. Sadly, the IC* debugger is currently unique in these respects.

2.4.9 State Vector Model

The state model is an abstraction through which the user perceives and manipulates elements of the target's state vector. The model is used to chunk and filter components of system state to increase user understanding of the dynamic structure of program data. Manipulation is supported to facilitate experimentation. State models have four main attributes:

- **Extent**, the breadth of coverage of the model as compared to the computational model of the target;
- **Level**, the level(s) of abstraction at which data is presented or manipulated;
- **Specifier**, the mechanisms provided to specify the desired subset of the state vector for a certain operation; and
- **Modifier**, the mechanisms available to the user to change its value.

Visualization, the means by which this specified facet of the vector is conveyed and related user interface issues are covered in Section 2.4.12.

These attributes are considered for a range of debuggers in Table 2.3. This table has as much to say by what it omits as by what it depicts. Many of the tools surveyed do not cater for state models in *any* documented form [BFM+83, RRZ89, SBN89, HC89, Els89, For89]. Although some of these are experimental tools, this omission is surprising and unfortunate.

The computational model of the target language (the column labeled *Language* in Table 2.3), or system, is considered in terms of paradigm (as defined in [Weg90]), level of parallelism and

Debugger	*Language*	*Level*	*Extent*	*Specifier*	*Modifier*	*Reference*
Jdb	seq, proc	high	all	name	expression	[WN88]
Dbxtool	seq, proc	high	all	expression, active	expression	[AM86]
PDF	seq, proc	low	all	address	value	[Car86a]
Exdams	seq, proc	high	all	name, time, dependency, range	value	[Bal69]
IDE	seq, proc	medium	all	name, active	expression	[Chu83]
Ups	seq, proc	high	all	name, active	value	[Bov86]
ThinkC	seq, proc	medium	all	name, active	value	[GAS+86]
VAX DBG	seq, proc	low	all	static path, active, expression	expression	[Int84]
PROVIDE	seq, proc	high	all	icon, time by assertion, active	expression	[Moh88]
DIS	con, O–B	high	all	dynamic path expression	expression	[BTM89]
H/T	con, proc real-time	high	all	dynamic path active, range	value, trigger	[Bem86]
DISDEB	con, proc	medium	inter-process	name, active	value	[PL86]
CBUG	con, proc	medium	intra- & partial inter-process	name, active	value	[Gai85]
Multibug	con, proc	low	inter-process	static path, time active	value	[CP86]
Pi	con, proc	high	intra-process	expression	expression	[Car86b]
Realbug	con, proc real-time	high	all	expression	value	[BEH88]
ConDbg	con, proc	high	inter-process	name, time dependency	value	[Sto88]
Track	seq, O–O	high	all	expression, active	expression	[BH90b]
ST80	seq, O–O	high	all	expression	expression	[Gol83]
O_2	seq, O–O	high	all	name, icon	value	[DP89]
GDB	seq, O–O	medium	partial intra-process	name, time active	expression, trigger	[Sta88]
Parasight	con, proc	high	n/a	n/a	n/a	[AG89]
Blackbox	con, proc	high	all	name, active assertion, time	expression, trigger	[GKY89]
O/A	con, formal	high	all	name	expression	[LL89]
DPD	con, O–B	high	intra-process	expression, time	expression, trigger	[HK89]
Igor	seq, proc	medium	all	static path, time	value	[Fel89]
IC⋆	con, formal	high	intra-process	dynamic path, time, range	value	[CC89]
Spider	con, proc	high	inter-process	name	value	[Smi85]
MAD	con, proc.	high	all	name, active		[RRZ89]
Voyeur	con, proc	high	intra-process	expression	value	[SBN89]
Pathrules	con, proc	high	all	name, dynamic path	expression	[Bru85]

Table 2.3: State Vector Model Support in Modern Debuggers

any special considerations, e.g. being based on a formal model (e.g. [BFV86]), or the requirement for real time operation (e.g. [Bem86, PL86]). Within the paradigm classification, *Proc* denotes procedural, *O–B* object based and *O–O* object oriented. The extent of parallelism is classified as sequential (*seq*) or concurrent (*con*).

The *level* of a debugging tool concerns the abstractions through which the tool allows the user to access and manipulate state. High level tools are those which support high level languages (e.g. Pascal, Ada) and offer only high level, and user created, abstractions to access and modify state (e.g. assignment to literals, user constructors, function calls). Low level tools are designed to support the assembly languages of specific microprocessors and offer only machine based abstractions (e.g. memory access, machine representations of data). Medium level tools are those which support high level languages with both high and low level abstractions. Although the latter is typically done for pragmatic reasons, it often undermines the computational model of the supported language(s) and confuses novice users.

The *extent* of a debugger (see the *extent* column of Table 2.3) depicts how much of the target state vector is accessible by the user through the debugging tool. Debuggers supporting concurrent targets can access two domains of that vector: the inter- and intra- process state, sequential targets have only the latter. Intra-process state consists of the values of all program internal variables and any expressions thereof; inter-process state includes objects such as mailboxes, queues, process flags, semaphores and shared memory, which are essential to the full understanding of a parallel system. As the table shows, not all debugging tools are able to access both domains of the target and on occasions, due to incomplete symbol tables, only partial domain access is implemented (e.g. [Sta88]). This is a severe and exasperating limitation for which there is seldom any good reason.

The *specifier* column describes the mechanisms used to specify a part of the state vector that is to be viewed, altered, or otherwise used by some debugger command (i.e. debugger *lvalues*). Often variables are identified (textually, or by selection) purely by name or icon (or address in low level systems). Many systems allow the value of variable expressions to be calculated. More advanced systems allow the user to specify a *pathname* which, in addition to variable name, specifies the nested invocation context of the variable concerned to avoid name clashes; *dynamic pathnames* allow the user to specify variables that are scoped within, as yet, non-existent stack frames. This allows the user to specify variables that may exist only after several levels of recursion have occurred, before execution has even started [Bem86]. Time may also be used as a specifier (to permit the viewing of past values of system objects), as may data flow dependency or the value range. A few systems (e.g. [Moh88]) combine these specifying constraints into an assertion language, allowing such commands as "*display the value of x before it last contained a number greater than 100*".

Under the specifier column we also indicate whether, or not, the object displays maintained by the debugger are *active*. That is, are the values depicted correct only at the time of the query, or do they pre-emptively alter to reflect current program state.

A *modifier* is a value to which a debugger may alter an variable within its extent (i.e. a debugger *rvalue*). Many debuggers are only able to assign the values of literal constants to specified variables. Those that have an in-built interpreter (e.g. [BH90b]), may assign the values of expressions (either in the target language, or in one unique to the debugger) to accessible variables. Still greater flexibility is offered by these tools which permit pre-programmed command sequences, including state modifying commands, to be executed when certain state predicates become true. These are called *triggered* modifiers (labeled *trigger* in Table 2.3). For example, the data path debugging system [HK89], as will shall see in Section 2.4.10, is built heavily on data-based predicates. DPD has an extensive set of debugger commands (including state alteration) that may be triggered when data objects change, or adopt certain values. These *immediate actions* are analogous to the syntax-directed translations of YACC [Joh78] and can be used to visualize, or even alter, program behaviour.

2.4.10 Behavioural Model

The behavioural model is an abstraction through which the user perceives and controls the run-time activities of her program. Because of the importance of these activities in achieving some understanding of the target, and in establishing contexts in which hypotheses about failure can be interactively tested, these aspects of a debugger's functionality are the most critical. Essentially, three types of behavioural model exist: the *lexical* model which defines behaviour as a list of source lines visited, or modules entered—on which most breakpoint models are defined; the *data event* (or data flow) model, which expresses activity purely in terms of state vector deltas; and the *control event* model, which maps behaviour on to a stream of primitive parameterized events. These ideas and their relation to some of the tools surveyed can be seen in Figure 2.3.

Table 2.4 contains a more expansive listing of the behavioural models of the tools surveyed, which includes the paradigm supported (using the same key as Table 2.3), mechanisms used in the model and the visualization and control features it supports.

Traditionally, debuggers have used a *lexical* behavioural model and many event-based tools retain one. Event models are classified by the alphabet of events they support. Control events (*C-event*) denote atomic actions associated with control flow: e.g. the entry or termination of a module, the sending of a message, or the use of a synchronization primitive. Data events (*D-event*)

Debugger	*Paradigm*	*Model*	*Visualization*	*Control*	*Ref*
Jdb	seq, proc	lexical	trace	cbrk, 1step	[WN88]
Dbxtool	seq, proc	lexical	trace	cbrk, 1step, nstep, exec	[AM86]
PDF	seq, proc	lexical	trace, T-filter	stop, brk, 1step	[Car86a]
Exdams	seq, proc	lexical	trace	stop, cbrk, 1step, trigger replay, reverse	[Bal69]
IDE	seq, proc	lexical	trace	stop, 1step, simulate	[Chu83]
Ups	seq, proc	process, lexical		brk, 1step, sig	[Bov86]
ThinkC	seq, proc	lexical	trace	stop, 1step, cbrk	[GAS+86]
VAX DBG	seq, proc	lexical	P-trace, E-filter	brk, nstep, stop amend, trigger, goto	[Int84]
PROVIDE	seq, proc	lexical, log, D-event	trace	brk, 1step, gait, goto replay, reverse, amend	[Moh88]
DIS	con, O–B	lexical, C-event log, process	trace	cbrk, simulate, stop	[BTM89]
H/T	con, proc real-time	lexical, C-event process, eventspec	trace	cbrk, stop, time	[Bem86]
DISDEB	con, proc	event, process	trace, E-filter	cbrk, time	[PL86]
CBUG	con, proc	lexical, process	trace	cbrk, 1step	[Gai85]
Multibug	con, proc	event	trace	cbrk, 1step, stop, goto trigger, sig	[CP86]
MuTEAM	con, proc	process, A-event eventspec	trace	cbrk, comp	[BFM+83]
Pi	con, proc	lexical, process	trace	brk	[Car86b]
EBBA	con, proc	process, eventspec	trace, cluster	comp, trigger	[Bat89]
Realbug	con, proc real-time	process event, log	trace, T-filter	cbrk, time, amend	[BEH88]
MAD	con, proc	event, log eventspec	cluster, filter	n/s	[RRZ89]
Voyeur	con, proc	lexical, A-event	trace	stop, 1step	[SBN89]
ConDbg	con, proc	eventspec, process, depend, D-event		brk, replay	[Sto88]
Amoeba	con, proc	lexical, eventspec C-event, process, log	trace, filter, cluster	brk, comp trigger, replay	[Els89]
TRACK	seq, O–O	lexical, log, C-event	trace, EP-filter	gait, brk, stop, 1step	[BH90b]
ST80	seq, O–O	lexical	trace	brk, stop	[Gol83]
O_2	seq, O–O	lexical, log	trace	brk, 1step, trigger, skip sig, time, amend, exec	[DP89]
GDB	seq, O–O	lexical, process	trace	stop, ncbrk, exec, nstep, trigger, amend	[Sta88]
Parasight	con, proc A-event	lexical, process	by user-defined s/w	by user-defined s/w	[AG89]
Agora	con, proc	event, process, log	cluster	cbrk, replay	[For89]
Blackbox	con, proc	lexical, event, log	E-filter trace	stop, brk, simulate, replay, comp, gait	[GKY89]
O/A	con, formal	C-event, time, log	trace, filter	1step, gait, simulate, reverse	[LL89]
DPD	con, O–B	event, process, log depend	trace, E-filter	brk, 1step, amend comp, trigger	[HK89]
Igor	seq, proc	D-event		amend, replay, reverse	[Fel89]
IC⋆	con, formal	D-event, time		simulate, replay, reverse amend	[CC89]
Spider	con, proc	C-event, log	trace, EP-filter	stop, 1step, event amend, comp, trigger	[Smi85]
Pathrules	con, proc	event, lexical eventspec	trace, filter multi window	amend, 1step, cbrk gait, trigger	[Bru85]

Table 2.4: Behavioural Model Support in Modern Debuggers

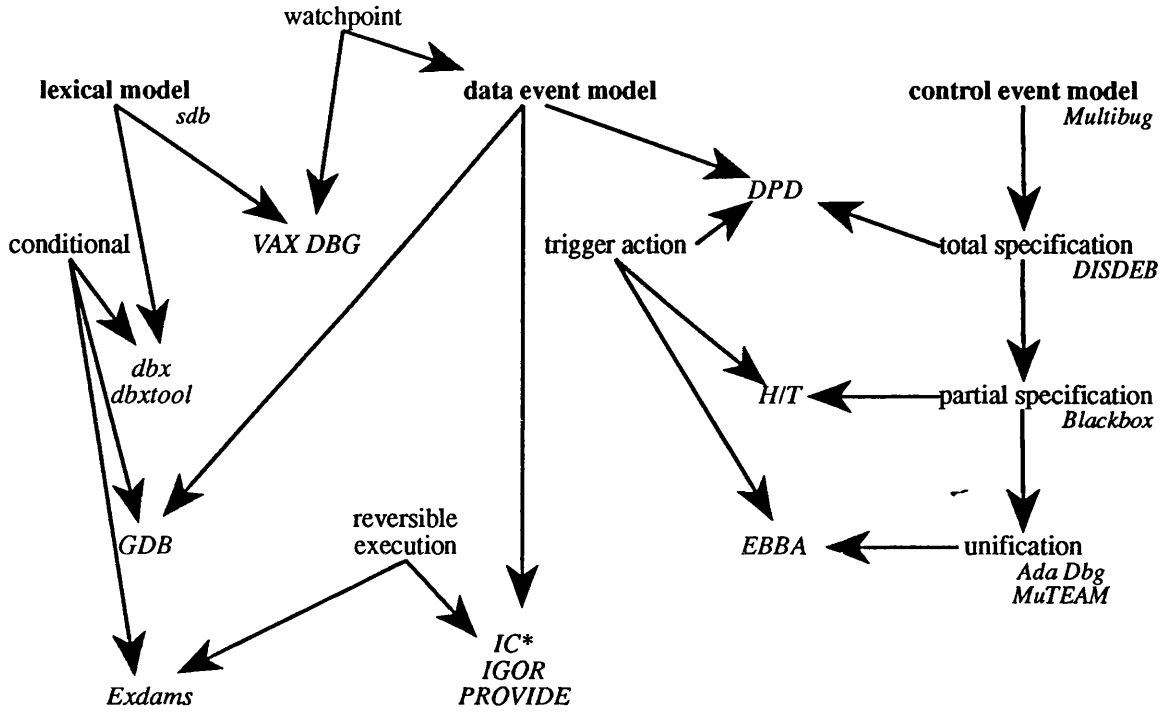

Figure 2.3: Relationship of Behavioural Models to Debugging Tools

represent atomic data actions: e.g. the access or alteration of a system object; some systems advance this, enunciating the data flow dependencies (labeled *depend* in Table 2.4) between events. Annotated events (*A-event*) are those which represent user-defined, in-program instrumentation and are typically used in program-animation systems [LD85, Bro88]. Some alphabets may contain all of these event types (denoted *event* in Table 2.4). Advanced systems may permit the user to hierarchically define new events from existing ones (*eventspec*), which helps to promote user defined abstractions (see Section 2.4.11). Furthermore, some models record a history of a program's execution (*log*) in an appropriate form to facilitate time based querying, or include detailed timing information (*time*) to facilitate the testing of real time programs. Debuggers for concurrent languages often support process-oriented events (*process*), allowing users to directly view and alter inter-process behaviour of the target.

Visualization, in the context of this thesis, refers to how program activity is relayed to an observer and does not imply any graphical (or other) representation for such a display. Here, we consider what information is relayed; in Section 2.4.12 we consider the media used to convey this information. The most common visualization technique is the *trace* (labeled *trace* in Table 2.4), a depiction of all activity at a fixed level of abstraction determined by the model. For example, a

lexical trace typically constitutes a list of all the source lines being executed. Advanced systems may enable this exhaustive (and often voluminous) display to be controlled by tracepoints (labeled *P-trace* in Table 2.4), which activate traces only after certain conditions have been satisfied. Alternatively, more general trace filtration and abstraction facilities may be offered including: filtering by entities of the model that are involved (*E-filter*), i.e. a procedure, source line or data object; filtering by the type of activity or instruction involved (*T-filter*), i.e. assignment, function call; and filtering by the parameters of actions (*P-filter*), especially relevant to event-based systems. Some filtering systems may combine all of these approaches (*filter*). Others may facilitate trace abstraction by allowing users to hierarchically define 'higher-order' events in terms of event sequences. This clustering (*cluster*) helps to reduce the volume of trace information whilst retaining all of the semantics. Any system which allows debugger command sequences to be triggered from specified conditions (*trigger*), obviates the need for specific tracing and filtering of behaviour.

The debugging mechanisms that facilitate target control are remarkably similar, in otherwise dissimilar debugging tools. They allow the user to manipulate the progress of program execution. The most important facility is that allowing the user to halt program execution, at a location (for lexical models) or after an event, to establish a context for examination and further experimentation. The traditional functionality of debuggers includes unconditional breakpoints (particularly lexical breakpoints) (*brk*); breakpoints dependent on some aspect of state (*cbrk*); breakpoints that become active after being activated a certain number of times (*nbrk*); executing single or multiple instructions (*1step*, *nstep*); skipping instructions (*skip*); interruption of execution by a keypress (*stop*); starting execution at a certain point (*goto*); executing external modules (*exec*); and the patching of the target run-time image (labeled *amend* in Table 2.4, see also below). Many of these facilities are merely translations of those offered by assembly language debuggers of the 1960's [Moh88, MPW89]. More advanced functionality includes: the ability to trigger sequences of debugger commands on exhibition of certain behaviour by the target (*trigger*, see Section 2.4.11), specified using the abstractions of the model; the ability to record execution and then replay, simulate, or even reverse it (*replay*, *simulate*, *reverse* see Section 2.4.13); manipulation of actual and simulated stimuli, external to the target, such as interrupts, signals and the apparent flow of time (*sig*, *time*); controlling the rate of execution (*gait*); and arguably the most powerful: the comparison of actual behaviour with a user provided specification (*comp*, see Section 2.4.11). The latter can be used to highlight behavioural deviations and support hypothesis confirmation.

The latter facilities more closely support the demands of the debugging task addressed in Section 2.4.5. The former, particularly lexical breakpoints, are *expected* in modern debuggers [Car86a] and yet they are a hangover from the assembly language era. They are easy to implement (typically, an instruction is substituted with an interrupt command), but often painful to use.

Although they can be used to establish a context, one must be aware in advance of the textual location of that context and once it is established one may be unaware how and why execution has progressed to this point. Furthermore, they offer a very limited perspective of program behaviour, fixed rigidly to line number granularity. In the next section we shall consider a better approach.

The ability to *patch* object code, i.e. to dynamically and ephemerally alter the target to circumnavigate a bug, is a somewhat misplaced facility in modern debugging tools. The initial need for patching—to amend the target to confirm the validity of an alteration, without the overhead of recompilation—has been obviated by the introduction of incremental compilers. Furthermore, patching object code from an assembler (which is how the technique originated) is only moderately dangerous, because the patch is at the same level of abstraction as the source—the same is not true of patches applied to object code from a high level language compiler. Consequently, one has nothing to gain by patching and a great deal to lose: the mismatch of source semantics and run-time behaviour is potentially disastrous [Car86a].

2.4.11 Event Based Models of Behaviour

Event based models of behaviour are becoming increasingly popular with debugging tools designers and are potentially, extremely flexible [CBM90]. Parameterized events are a better basis for a model because: they allow a 'natural', hierarchical expression of behaviour [Sen83]; provide a heterogeneous, abstract specification medium free from the vagaries of any one language or system [Bat89]; and offer a wide range of abstraction levels [Bat87a, LL89]. The latter is particularly important since abstraction is the key to managing the complexities of debugging [Bat89, Els89] and can help users to express behaviour at the level of the problem domain—unlike lexical models.

Event based systems have four important properties which can be used to classify them: the alphabet of events (and event parameters) used to describe behaviour; the composition mechanisms by which primitive events are compounded into specifications of complex actions at the problem's domain of abstraction; means of constraining specifications to define their coverage; and the range of activities that may be *triggered* by the failure or success of a specification [Bat87a]. In Table 2.5 we consider these properties for the event-based specification systems covered by the survey.

The alphabet of primitive events (and their parameters) used to express behaviour characterize the system for which they are defined and should be chosen with great care [Bat87a, LL89, BFM+83, RRZ89], especially in tools which enable the comparison of monitored events with a user specification (e.g. [Bat89]). It is desirable to invent an event alphabet customized for the target paradigm [BTM89], with equal emphasis on control and data events [Bal69]. Primitive events may

Debugger	*Paradigm*	*Event Alphabet*	*Composition*	*Constraints*	*Triggers*
DIS	con, O–B	informal, 28, ada specific	single event	unified	
H/T	con, proc	informal, 13, in 4 classes	seq, conj, disj partial		breakpoint start trace
DISDEB	con, proc	informal, memory & port i/o	seq, conj, disj parallel		any debugger command
MuTEAM	con, proc	formal & user def'd CSP based	seq, rep, conj, disj, partial	unified, boolean	
EBBA	con, proc	unspecified & user def'd	seq, rep, neg, conj, disj, parallel, partial	boolean event/param filter	
Realbug	con, proc	n/s	single event		
Voyeur	con, proc	informal, annotative & user def'd	single event		
Amoeba	con, proc	informal, comms, IPC, sys call & user def'd	seq, rep, disj, neg parallel, partial	event filter	any debugger command
Blackbox	con, proc	informal, 5 primitive & user def'd	conj, disj	boolean, temporal logic event/param filter	
O/A	con, formal	informal, annotative, 4 IPC events	single event	event/param filter	
DPD	con, O–B	informal data access, IPC	seq, rep, disj, conj, parallel	event filter	any debugger command
Spider	con, proc	informal, IPC & user def'd	single event	boolean event/param filter	any debugger command
Pathrules	con, proc	4 informal, IPC data & user def'd	rep, seq, parallel conj, disj	boolean	see/alter client pathrule or environment
MAD	con, proc	informal annotated data, IPC, h/w user def'd	as Pathrules	boolean	

Table 2.5: Event-based Specification in Modern Debuggers

be system generated, by fixed instrumentation, or added by user annotation as in [SBN89, RRZ89]. The latter supports application specific events and is certainly easier to use, but at the cost of completeness, consistency, time and efficiency. Such ad-hoc annotation is akin to program amendment, and potentially shares all the disadvantages. As Table 2.5 shows, all of the surveyed debugging tools that incorporate event-based specification systems, use informally generated event alphabets, except *MuTEAM* [BFM+83]. These alphabets vary considerably and many allow user defined, (source annotated) events to be added to the primitive set (labeled *user def'd* in Table 2.5).

To support abstraction, most of the tools supporting event-based specification provide a language with which to generate composite events from primitives. Typically this language enables complex events to be composed of sequences (*seq*), concurrent conjunctions (*conj*), disjunctions (*disj*), concurrent shuffles[9] (*parallel*), repetitions (*rep*) and negations (*neg*) of primitive events (or lesser complex events). Advanced systems allow partial specification (*partial*) and, in some cases,

[9] An operator specifying a list of events that may be executed, concurrently, in any order.

the partially specified sections may be unified (*unified*) with the events they match in a proven specification and the resultant mapping used in constraints. Constraints strengthen a specification medium otherwise completely reliant on pattern matching. They can be defined as simple filters on event type, or parameter values. Boolean expressions and even temporal logic equations are provided by some systems. There is also massive variation in what actions debugging tools can trigger if these specifications fail.

The abstraction and complexity of complex events is such that some systems (e.g. *EBBA* [Bat89], *Amoeba* [Els89] and *SPIDER* [Smi85]) offer library facilities to promote the storage and reuse of event specifications [CC89]. There are great difficulties indexing these specifications such that they can be efficiently recalled later for reuse. Furthermore, no system allows existing specifications to be parameterized or strengthened by new ones.

The ability to compare a behavioural specification with the monitored behaviour of a program is extremely useful debugging aid. As we shall see in later sections, it is especially useful in debuggers for concurrent programs. Naturally this technique is not a panacea, the user's event specification may itself be flawed. However, it does offer a potentially vital second opinion, from an operational viewpoint quite different from the programmer's. It is surprising that so few debugging systems support this functionality, and of those that do, that so few support it well.

2.4.12 Human–Computer Interface

The user interface of a debugging tool has two main requirements: to manage the complexity of the debugging tool by providing the user with an easily remembered interaction dialogue and command structure (since most debugging tools are only used intermittently); and to allow the user to interact with the three models of her program in order to attain a deeper comprehension of it. This leads to a split in user interface resources between control of the debugger and visualization of the target. The facilities used by debugging tools to support user interaction are listed in Table 2.6. As the table shows, some debugging tools offer no user interface support because they are not intended for interactive use.

The many different context dependent views that are required by the users of debugging systems (see above) and the inherent complexity of these views makes debugging tool, user interface design extremely demanding [BEH88]. Modal interfaces lack the required flexibility [Car86b], but single stream non-modal designs (such as those based on textual terminals) are woefully under-powered and often confusing to use. Interaction through a set of windowed, virtual terminals (labeled

Debugger	*Interface Modalities*	*Source Visualization*	*State Visualization*	*Behavioural Visualization*
Dbxtool	cli, window, menu, prog	text	text	text, highlight
Jdb	cli, window, menu	text	text, adapt, customize	
PDF	window, menu	text	text, adapt, customize, user def'd	object text, customize
Exdams	cli	text user def'd	text, analog customize	text, graphic customize
IDE	cli, save	text	text	text, analog
Ups	window, menu	text	text, d-manipulate	text
ThinkC	cli, window, menu	text	text	text
VAX DBG	cli, prog	text	text	text
PROVIDE	window, menu	text	analog, graphic, customize user def'd, d-manipulate	text, graphic highlight
DIS	lang, cli	text	text, graphic, user def'd	text
H/T	window, lang, menu	text	text	text
DISDEB	cli		text	text
CBUG	window, menu	text	text	text, highlight
Multibug	cli	text	text, user def'd	text
MuTEAM	cli	text	text	text
Pi	window, menu	text	text	text
EBBA	n/s			text
Realbug	lang, window, menu	text	text, analog	text
MAD	window, menu		graphic, user def'd	
Voyeur	window, menu		text, graphic, user def'd	
Belvedere	window		n/s	graphic
ConDbg	cli, window	text	text	text
Amoeba	n/a	n/a	n/a	n/a
TRACK	window, menu	text	text, user def'd, graphic	
ST80	window, menu	text	text	text, highlight
O_2	window, menu	text	graphic, d-manipulate	text, highlight
GDB	lang, cli	text	text, customize	text, customize
Parasight	cli	text	text, customize, user def'd	text, user def'd
Agora	n/a	text, customize		
Blackbox	n/a	text	text	text
O/A	cli, window, menu	text	text	text
DPD	cli	text	text	text
Igor	n/a	n/a	n/a	text
IC⋆	window, cli	graphic, analog, text	adapt, graphic, analog text, d-manipulate	adapt, graphic, text
Spider	cli	text	text	text
Pathrules	cli, window	text		text

Table 2.6: The User Interface Facilities of Modern Debuggers

window in Table 2.6) partially solves this dilemma, as does the use of hierarchical menus [BEH88] (*menu*). Despite this, most debuggers still use standard command line interfaces (*cli*).

The chief problems of debugger control are: how does the user specify which aspects of her program she wishes to see; how is this information presented in order to avoid overwhelming her and how is mastery of the tool made both easy to attain and remember. The current consensus is that command interfaces should be orthogonal and reflect the abstractions of the target language as much as possible, making the commands easy to remember and allowing the user to specify what she wishes to see using these abstractions [BEH88, Bem86] (such systems are denoted *lang* in the *Interface Modalities* column of Table 2.6). Alas, few debuggers attain this ideal. Debugging session management is supported in some debuggers by the use of session checkpointing [Car86a, Chu83] (*save*), programmability [Sta88] (*prog*) and integration with software development environments [BH90b, Gol83].

The appropriate graphical visualization techniques can make debugging tools aid the programmer, as effectively as analogical hardware tools (e.g. the oscilloscope) help an engineer [BH90a]. It is frequently argued that patterns of behaviour are best understood visually—especially through animation. The advantages offered by graphical media (labeled *graphic* in Table 2.6) is that through panning and feature size one can create highly selective displays with variable degrees of emphasis on each feature. In contrast, text emphasizes everything to the same degree (notwithstanding techniques like highlighting, colour and font, which can easily be saturated [Mye84, Bal84, Shn87]). The effectiveness of a display is inversely proportional to complexity [Knu63]; thus selective displays are essential to avoid overwhelming users with too much information [DC86].

What can be visualized depends on the models supported by the debugging system, the animation techniques supported by the debugger and, in an event-based system, the alphabet of events used. Many systems require the user to provide her own visualizations (labeled *user def'd* in Table 2.6) and animation rules [RRZ89, SBN89] (*customize*). While this is a flexible technique, the defining mechanism should be easy to use, and enable the reuse of a host of pre-defined graphical visualizations. Given the reluctance of most people even to use a debugger, custom visualizations will be of interest to a small minority of users. The very best visualizations have some innate similarity with that which they represent. Such analogical visualizations (labeled *analog* in Table 2.6) are notoriously difficult to design. Enhancing this analogy still further, there are systems which allow visualizations to be directly manipulated (*d-manipulate*), such that changes in their visual structure result in analogous changes being wrought on the entity they represent. An excellent survey of visualization techniques can be found in [DC86].

It is a considerable shame that the majority of debugging tools do not attain these ideals. It

is also disconcerting that many debuggers which claim to provide graphical visualization, do not[10] [DC86, DP89]. Of the debuggers surveyed, only four (IC⋆ [CC89], O_2 [DP89], Track [BH90b] and PROVIDE [Moh88]) are capable of analogical, graphical state visualization of the quality pioneered by Myers [Mye83]. Several displayed a potential to visualize behaviour using an event based model [Bat87a].

The potential of graphical visualization is vast, but bounded. Visualization only relays the effects of execution, not the causes. Algorithms can not be directly visualized, only the consequences of their execution. Thus, it provides understanding of programs' effects without necessarily relating it to causes in the program. Although there are many claims that graphical debugging improves programmer productivity (e.g. [BEH88]), no experimental evidence exists to prove this assertion.

2.4.13 Problems Introduced by Parallelism

The introduction of parallelism greatly complicates debugging [BTM89, Bat89, For89] chiefly because: the asynchrony of parallel systems makes their behaviour more difficult to understand [RRZ89, SBN89, HC89]; parallel systems can fail in more intricate ways such as timing errors, deadlock or starvation [GKY89]; and an overwhelming array of information needs to be digested before the state of a distributed program can be appreciated. Ironically, these problems mean that the need for debugging tools in parallel systems is greater than that in sequential systems. Thus far, this need has been poorly served [For89].

The problems of parallelism can best be viewed by analyzing the assumptions made for sequential systems which are no longer valid [BFM+83]—these include those listed below.

- **Determinism.** The non-determinism of parallel programs is a severe handicap to repeatable execution which is essential to hypothesis confirmation in debugging. As a result, a host of mechanisms have been invented which attempt to assert determinism.
- **Debugger Functionality Independent of Architecture.** Although this can be guaranteed in most sequential systems, parallel architectures show much more diversity. This diversity includes fundamental concepts that effect debugging behaviour such as synchronization constructs, means of inter process communication and the granularity of parallelism. This means that debuggers are less portable in parallel environments.

[10] There is a temptation to state that a debugging tool is graphical merely because it supports bitmapped windows.

- **Debuggers May Alter the Target's Time Characteristics With Impunity.** Such alteration, usually as a side effect of code instrumentation or a deliberate action to slow or halt a program for debugging purposes is, at worst, a matter of mere inconvenience in sequential systems; in parallel systems any perturbation of inter process timing can completely alter system behaviour. The *Probe Effect* [Gai85], as this is known, is a serious limitation in parallel debugging.

- **One Focus of Interest.** Parallel systems embody multiple threads of control. Consequently some powerful user interface techniques will be needed to ensure the user is not overwhelmed by information, does not miss vital details and is able to co-assimilate information from multiple concurrent sources.

We consider some of these points in greater detail below. Some of them constitute such problems, that no single debugger can provide a complete solution and retain all the functionality traditionally associated with sequential debugging. This has prompted the development of 'multi-stage' debugging models [LL89, Smi85] in which traditional debugging tools are used to debug each process on a stand-alone basis (an *intra-process* tool), and then the processes are composed and debugged by tools which deal only with *inter-process* abstractions [BFM+83]. Some argue that this approach is too modal, or has too high an overhead, and that such functionalities should be combined [BTM89, Els89, Car86b, Bat89]. Others argue that, like the architectures of the host systems, a debugging tool should be distributed [Bat89].

Non-Determinism

Repeatability of program behaviour is an essential part of debugging, just as experimental reproducibility is vital to scientific method. A continuum of methods exists to enforce repeatable behaviour by *replaying* programs: spanning from full simulation, to event-driven, constrained re-execution. These include those listed below.

- **Statistical Re-execution**, unconstrained re-execution of a non deterministic program with sufficient frequency as to statistically ensure the eventual replay of previously observed behaviour. This, somewhat exhaustive technique, was first formally suggested by Gait and used in his CBUG debugger [Gai85].

- **Guided Re-execution.** The program is re-executed under constraint. That is, normal execution is allowed to proceed until some juncture at which inter-process communication,

or a non-deterministic choice is to be made, then the program is constrained to behave as previous executions did. Examples include Stone's speculative replay and Miller and Choi's incremental tracing mechanism based on flowback analysis [Sto88, MC88]).

- **Full Simulation**. The program run is fully simulated, by-passing any non-determinism. For example, Kepeklian's DTML debugger [Kep87]).

Gait's method aside, all forms of re-execution depend on the generation of history logs during an initial execution of the errant program [LL89]. The concept of history tapes is not new—they were introduced by Balzer in the Exdams debugger [Bal69]. Furthermore, they have many other uses; for example, in Blackbox [GKY89], they are used to maintain a history of execution which can be queried, later, with temporal assertions. Ideally, such logging would always be enabled, allowing impromptu debugging sessions. Unfortunately, this is usually infeasibly expensive, and consequently, non-determinism still constitutes a problem if an instrumented re-run of a program behaves differently from the original.

The form and extent of history logs varies with the replay method employed. Logs are often parameterized event lists. Those logs generated to facilitate simulation must represent *all* of the details concerning the run-time behaviour of the program. Such exhaustive logs are often enormous (the efficiency of history logs is discussed later in this section). By comparison, logs for guided re-execution merely have to provide partial details concerning process timing and dependencies (i.e. record all the non-deterministic choices), such that determinism can be asserted over the re-run program to provide repeatability. This is achieved by executing the program under a supervisor process and using breakpoints to block for inter-process dependencies. Often, execution order does not have to be fully specified and a partial record is sufficient—knowing when the order is significant is a problem in itself. Consequently, the logs are reduced in size, as is the level of control over the re-run program. The disadvantages of simulation are offset by the fact that it does not require specialist hardware and one can experiment in ways that are otherwise impossible, for example: reverse execution.

Dependence of Debugging Tools on Architecture

Replay mechanisms are a good illustration of how debugger requirements vary heavily with the degree of parallelism, frequency and style of communications and type of architecture. How replay may be achieved is very dependent on these factors. LeBlanc and Mellor-Crummey [LMC87] achieve it by versioning shared variables and delaying variable access until its version is ready during

replay. Elshoff [Els89] re-uses this mechanism in his debugger for Amoeba. However, due to an asynchronous construct unique to Amoeba, the method is not wholly effective [Els89]. Forin [For89], also uses versioning, but his implementation is more restrictive and does not support the concurrent read exclusive write (CREW) protocol because of the latter's inefficiencies; indeed, under some circumstances, it uses write-once memory instead (thus reducing size of event history log). Stone [Sto88] uses a *speculative* replay mechanism, which detects instances in which replay sequences violate a recorded dependency. In such a case, the scheduler is rolled back and another event sequence is tried. SPIDER [Smi85] works in a non memory sharing environment and consequently must record and replay behaviour guided solely by logs of inter-process communication.

The degree of dependence of a debugging tool is the range of external agents it relies on and the extent to which this reliance causes changes in structure of the agent to necessitate changes in the debugger. These agents include, but are not limited to those listed below.

- **Hardware.** Low level debuggers, like PDF and AFD [Car86a], directly support the instruction set of the microprocessor on which they run and are thus dependent on it. Many high level debuggers for parallel systems rely on ad-hoc hardware alterations (or additions) to achieve some desired functionality, or to provide sufficient efficiency to reduce the probe effect. For example, DISDEB [PL86] avoids software and kernel dependencies by using custom-built bus monitors to ascertain the status of nodes in a distributed system. Bemmerl's real time debugger for host/target environments [Bem86] and the MAD [RRZ89] debugger use a similar scheme. Generally, hardware dependencies are more prevalent in debugging tools for parallel or real-time systems.

- **Kernel.** Some debugging tools utilize custom kernel alterations to acquire the necessary efficiency, for example MuTEAM [BFM+83] and Multibug [CP86]. Others use it to achieve functionality which would be unattainable otherwise, e.g. the incremental, page-based checkpointing of IGOR [Fel89] which supports its rollback mechanism. Other tools reply on the facilities of a particular operating system to such an extent that they cannot easily be ported to another. Elshoff's Amoeba debugger [Els89] has a replay facility that is severely weakened due to the design of the host operating system. Similarly, CBUG, Gait's prototype C debugger [Gai85], is heavily dependent on UNIX.

- **Compiler.** Like hardware dependencies, reliance on compiler internals is confined largely to parallel debuggers. It is often caused by implementation of the instrumentation required to achieve adequate behavioural information. For example, Bemmerl's real time debugger relies on an instrumented compiler to output event parameters to the custom hardware discussed

above [Bem86] and IGOR [Fel89] works with a compiler which has been altered to make internal data structures easier to access. Brindle et al.'s Ada debugger [BTM89] uses a custom compiler to facilitate programming at breakpoints.

- **Language**. Many sequential debuggers are specifically targeted at one programming language, providing functionality uniquely relevant to that language. Often, such tools are dependent enough on the abstractions of their language as to compromise the ease of porting the tool. Examples include: TRACK, which is deeply entwined with Smalltalk80 [BH90b]; the ThinkC debugger which cannot even be used with other variants of the C language [GAS+86]; and PROVIDE which supports its own subset of C [Moh88]. Parallel debuggers are *more* language dependent than sequential ones, due to the greater diversity of parallel languages. Synchronization schemes, communication mechanisms and the grain of parallelism are all factors which can change radically between concurrent languages and on which debuggers depend [BFM+83, BEH88, RRZ89]. This limitation is addressed to some extent by multi-lingual debuggers [ST83]. In practice, these are implemented by supporting a series of different, but analogous, languages through a flexible user interface and language independent code and symbol table structures, e.g. [Car83a]; or by encapsulating modules of different languages with a communications interface which makes them appear homogeneous [RSHWW88, For89]. Event-based models also help to reduce the language dependency of parallel debugging tools [Bat89, LL89, HK89].

- **Paradigm**. The paradigm [Weg90] of a language, or set of languages, is such a fundamental concept that no debugger is independent of it. The author is unaware of any debugger which supports multiple languages of a different paradigm, and doubts that such a tool would ever be feasible. Debugging is paradigm specific [DC86]: as is the choice of debugger functionality.

The advantages of debugger dependencies are discussed above. The drawbacks are largely unrelated to the type of dependency and include: limitation of distribution (a major reason why debugger usage is not more widespread, see Section 2.4.4); increased expense of development and maintenance—especially if the debugger must be altered whenever a new kernel is produced [Els89, Bov86]; and limited scope of application. The dependencies of the debuggers surveyed are shown in Table 2.7[11].

[11] The reader should beware that this table reflects those dependencies *admitted* by the authors of the works concerned. Where such dependencies were not discussed, or could not be inferred from details of the implementation, it is assumed—optimistically—that they do not exist. Table 2.7 consequently represents a best case for dependencies and the reality of the situation may be somewhat worse.

Debugger	*Dependencies*	*Reference*
Jdb	kernel, paradigm	[WN88]
Dbxtool	kernel, paradigm	[AM86]
PDF	hardware, kernel, paradigm	[Car86a]
Exdams	language, paradigm	[Bal69]
IDE	language, paradigm	[Chu83]
Ups	kernel, language, paradigm	[Bov86]
ThinkC	kernel, compiler, language, paradigm	[GAS+86]
VAX DBG	compiler, language, paradigm	[Int84]
PROVIDE	language, paradigm	[Moh88]
DIS	compiler, language, paradigm	[BTM89]
H/T	hardware, compiler, paradigm	[Bem86]
DISDEB	hardware, paradigm	[PL86]
CBUG	kernel, paradigm	[Gai85]
Multibug	hardware, kernel, paradigm	[CP86]
MuTEAM	kernel, language, paradigm	[BFM+83]
Pi	kernel, compiler, paradigm	[Car86b]
EBBA	kernel, paradigm	[Bat89]
Realbug	hardware, compiler, language, paradigm	[BEH88]
MAD	hardware, compiler, paradigm	[RRZ89]
Voyeur	paradigm	[SBN89]
ConDbg	paradigm	[Sto88]
Amoeba	hardware, paradigm	[Els89]
TRACK	language, paradigm	[BH90b]
ST80	language, paradigm	[Gol83]
O_2	kernel, compiler, paradigm	[DP89]
GDB	kernel, language, paradigm	[Sta88]
Parasight	hardware, paradigm	[AG89]
Agora	Language, paradigm	[For89]
Blackbox	paradigm	[GKY89]
O/A	kernel, paradigm	[LL89]
DPD	paradigm	[HK89]
Igor	kernel, compiler, language, paradigm	[Fel89]
IC⋆	hardware, kernel, paradigm	[CC89]
Spider	kernel, paradigm	[Smi85]
Pathrules	compiler, paradigm	[Bru85]

Table 2.7: Dependencies of Modern Debuggers

Latency and Time

The inter-dependent nature of distributed parallel systems compromises some traditional debugging techniques and reduces the effectiveness of others. Acquiring a consistent snapshot of the global (distributed) state is very difficult [Bat89, Els89] due to latency and the perpetual evolution of such systems. A related problem is that of breakpoints. Partial breakpointing of parallel systems is hampered by inter-process timeouts. A process has no *a priori* way of checking whether a process on which it is waiting has legitimately timed out or is breakpointed [Els89]. This causes breakpoints to perturb the behaviour of processes dependent on that which was breakpointed. RPC based systems do not have this problem, since within a program, process dependencies are explicit. However, external timeouts still pose a dilemma. Complete breakpointing of a distributed system might overcome these difficulties, were it not for the fact that it cannot be done *instantaneously* and thus the above problems are introduced again. Process logical clocks over which user has control is one solution [For89], but these are inefficient and dependencies must be calculated [AG89].

Monitoring the parallel behaviour of distributed systems (for replay or other purposes) is also fraught with problems. In order to determine the temporal order of the events recorded, events need to be timestamped. Maintaining any mutual consistency between these timestamps is hindered by the lack of a globally consistent clock. This can be partially solved through the use of a Lamport clock [LL89, Lam78].

Efficiency

Efficiency is an especially important issue in the design of parallel system debuggers, not least because of the probe effect (see Section 2.4.13). The surveyed systems use a combination of architecture and special techniques to improve efficiency—usually at the cost of dependencies (see Table 2.7).

Two activities notorious for their processing demands are scanning symbol tables (which can be very large in non-trivial programs) and generating an event-based history log (see *Non-Determinism* above). The first is usually overcome by using lazy evaluation of symbol table nodes to avoid the overhead of exhaustive strict evaluation [AM86, Car86b, Sta88] which is rarely required during normal debugging sessions.

The time and space inefficiency of event capture is a more problematic issue. Event capture is inherently very intrusive, especially data events, and this had led some designers to exclude such events from their log alphabets [BTM89]. However, data events are a powerful formalism

[LL89, HC89] and many seek to retain them and to try and reduce their inefficiencies. Many debuggers for distributed systems are themselves distributed [AG89], or are implemented such that they can access the multiple constituents of most programs (i.e. run-time image, scheduler, symbol table) in a decentralized way [Car86b]. Object oriented implementations are naturally inclined this way. Hierarchically distributed event-based systems can often distribute event capture and thus improve efficiency [Bat89]. This distribution has another advantage: it separates the address spaces of a debugger process from the target, preventing any corruption from rogue processes during debugging [Els89, Bru85, Smi85]. Efficiency can be further improved if only a minimal set of events are recorded and, at replay time, this program behaviour is dynamically reconstituted by static analysis and demand driven extrapolation of detail. This is termed *lazy-tracing*. An example of this is Miller and Choi's incremental tracing [MC88]—a replay technique so optimized through the use of symbolic, static dependence and data flow analysis that its tracing mechanism remains permanently active by default (see Section 2.4.13). The instrumentation of incremental tracing segregates problems into emulation blocks which are individually port trapped (events are generated on thread entry and exit to the block) with parameters concerning which data it is dependent on and which it alters. During replay, these blocks can be selectively re-executed to enhance the available information on the running program without the expense of executing them all or storing all the intermediate values otherwise needed. Block size is tailored to the host system. Trace overheads of less than 15% can be achieved in this way.

The space efficiency of history logs is also important, especially in systems which support full simulation. Such exhaustive logs are often enormous and techniques exist to restrict their size [Lar90]. Hardware is the ultimate answer to both space and time efficiency; in [RRZ89] an auxiliary processor is used to achieve monitoring in a shared memory environment. In addition, the MAD processor also performs event clustering, filtering and graphical visualization. A similar, if more dynamically configurable approach is adopted in [AG89].

Some argue that, to enhance efficiency, run-time monitoring (to maintain a history log) should be removed once programs are released after testing [Els89]. However, this can be dangerous because to remove monitoring after testing constitutes a change which might reveal bugs previously hidden by the probe effect and to disable the history log prevents 'impromptu' debugging sessions [ST83]. One solution is to make monitoring so computationally 'cheap', that permanently enabled logging is feasible [For89] (see above).

Visualization

Visualization of parallel behaviours and design of the user interface modalities required to support concurrency is also a significant problem. To avoid overwhelming the user, debuggers need to selectively examine behaviour, starting with an outline of the program behaviour and then steadily increasing the degree of detail as the user focuses on the bug [BH90a]. Thus, systems have to be (unobtrusively) monitored and then their behaviour visualized using a technique with a large capacity for abstraction. The facts that parallel systems introduce elements of behaviour of far lower level that those encountered in sequential systems (e.g. synchronization, time) and that debuggers need to analyze this behaviour at varying levels of abstraction, make a good case for using event based models of behaviour [BFM+83]. "Parallel algorithms are best understood in terms of patterns of IPC events" [HC89]. To avoid overwhelming users with a surfeit of information, the filtering and clustering operations that events facilitate are essential [Els89, For89]. Event based systems may also be used to compare real and intended behaviour [Bat89, HC89]. The event alphabet will be influenced by architecture: debuggers for non-shared memory systems, such as Belvedere [LMC87], will use alphabets based on inter process communication [HC89], as opposed to the versioning events used by shared memory system debuggers like Amoeba [Els89].

Many proposals exist to support the graphical visualization of concurrent behaviour [Fid89, Sto88, RRZ89, Agh90]. Graphical approaches are even more necessary for parallel systems, as simple textual traces would generate far too much information [RRZ89]. The most difficult problems in this regard are: representing the temporal order of processes; depicting dependencies; showing potential concurrency; and ensuring complete visualization. Basing a visualization on a formal model improves its rigour, e.g. Stone [Sto88] bases her visualization on a data structure which represents dependencies between concurrent processes. Like the more general concurrency model of Voyeur [SBN89], this ensures that the visualization is independent of language and operating system.

2.4.14 Problems Introduced by Object Orientation

The object oriented computational model is considerably different from the procedural norm [Mey88, Coo86, GR83, Str88], and although some similarities may be cited, the paradigm gap is wide enough to be considered significant by debugging tool designers. These differences and the ramifications they have on debugger design are explained in [PW91a] and summarized here.

Object oriented systems have many features which act to prevent bugs. The active separation of object signatures from implementations helps to reinforce the programmer's internal model of

her code. The encapsulation and *locality* [Lis87] of the paradigm reduce inter-object dependencies and make remaining dependencies explicit. This eases the location of bugs and localizes the effect of any (bug-ridden) code alteration. Unfortunately, this locality is somewhat compromised by inheritance and parallelism.

The object oriented paradigm brings new problems caused chiefly by inheritance and dynamic binding. The former, especially multiple inheritance, allows bugs to be propagated down the inheritance hierarchy and allows them to manifest themselves far from where they reside. Semantic interaction between methods co-inherited from different classes may easily result in objects with methods which use their representation in conflicting ways. Dynamic binding, as supported by techniques such as virtual functions (in *C++* [Str86]), as its name suggests, is not tractable to checking at compile time and can make programs very difficult to understand. Language environments that fully support dynamic binding (*LISP, Smalltalk80*) consist of mappings of names to values. Consequently, predefined functions can alter in behaviour unpredictably because names internal to their definition have been unwisely reused.

Debuggers for object oriented systems should accentuate the object model of the language for which they are designed; some debuggers based on hybrid object oriented languages that have evolved from procedural languages [Sta88] overlook this simple fact. Furthermore, many fail to use any object oriented abstractions in the models they present to the user [DP89, BH90a]. A visualization model of object oriented program execution has yet to be agreed. Opinion differs as to whether data and behaviour should be combined in a debugger's visualizations [BH90a] or separated [LL89].

2.5 Conclusions

In this survey we have compared and contrasted a representative sample of exception handling and debugging tools. We have defined what constitutes such systems, provided justifications for their usage, explained what commonalities they have and some of the ways in which current systems are still deficient. This thesis concentrates on the common problems of formality and abstraction.

The event-based model of behaviour is obtaining growing acceptance as the most able means of describing the behaviour of concurrent distributed systems. It is being used as a means of recording, visualizing and specifying such behaviour. However, no attempt has yet been made to formalize the alphabet of events used to express the behaviour of any particular system, or to permanently associate event-based specifications with the code they concern to prevent the

constant need to reproduce them. In addition, given the importance of the comparison of desired and actual behaviour during program debugging, few debuggers directly facilitate it. Finally, few debuggers for object oriented languages offer any direct support for object oriented abstractions.

Exception handling systems are also becoming increasingly accepted. Like debuggers, they seek to detect and locate bugs (although they have other functions). Such bugs are detected by distributing state specifications within programs. These specifications are of limited formality and applicability in parallel or object oriented systems. We have established the need for a behavioural specification medium to enhance formality and express the desired behaviour of concurrent objects.

In the remainder of this thesis we hope to explain how we have overcome the above problems by integrating some of the functionality of debugging tools and exception handling mechanisms, while enhancing the formality of both.

Chapter 3

An Operational Model of Object Oriented Systems

3.1 Introduction

Two issues raised in chapter 2 are addressed specifically in this chapter: the need for a structured, formal model to facilitate reasoning about specification media for parallel, object oriented languages; and the need to deduce the minimal alphabet required for such a specification medium, to minimize the bandwidth of the event stream representing behaviour. These goals are related: a minimized event alphabet can be derived from a consistent model—this is, in essence, the purpose of this chapter. We briefly recap on the motivations for building a formal model of object oriented systems here, before describing the model itself.

The advent of more sophisticated programming languages, caused by the recent popularity of the object oriented paradigm [Coo86, Mey88, Cox88b, Cox90, BGM89], brings an increasing need for formalisms that are capable of expressing the structure and behaviour of objects. This need is most acute where parallel object oriented languages [RWW88a, Ame87, Ame89a, CBM90, Hew77, Mod79, NH86, NP90] are used to harness multi-processor systems, because of the additional complexity involved (see chapter 2). Such formalisms could, if incorporated into programming languages, act as *life-script* mechanisms and be used to define or convey an object's process-based and state-based behaviour. This might constitute the basis of an in-source specification technique to check the correctness of run-time behaviour, constrain the available parallelism [Car89] or act as the basis of a 'semantic' browser. However, before such a formalism can be devised, a object oriented model is required as a framework.

One of the advantages of the object oriented conceptual model is the apparent elegance with which it can harness parallelism in multi-threaded systems. There is an inherent analogy between encapsulated objects, and inter-object message passing, and processes and interprocess communication [Car86b, RWW88a]. This structural isomorphism is bidirectional, implying that some of the existing formalisms and theories which have been used to define and evaluate process-based systems, are relevant to their object oriented analogues. A process calculus can, for example, be used to model object oriented systems. This would allow the full analytical power of such a formalism to be utilized in order to reason about the behaviours of such systems.

The goal of this work is to construct an *operational*[1] model of a general purpose, parallel, object oriented system. Also, we seek to ascertain the fundamental set of event classes that constitute its observable behaviour. The definition of this event *alphabet* is an essential primary stage in defining a formalism [LL89] that may be used to express object behaviour. Our model is built on the CSP process calculus [Hoa85] and uses a subset of the Z notation [Spi89] to express some aspects of state.

In the next section we explain why we chose CSP and list the salient features of a parallel object oriented system. In Section 3.3 the model is detailed, starting with an overview and a description of inter-object communication, before considering internal matters such as encapsulation, inheritance and the modeling of internal state. A detailed formal analysis follows in Section 3.4. This includes some discussion of the model's alphabet. The remaining sections: 3.5 and 3.6 cover the applications and limitations of this work.

3.2 Design of the Model

3.2.1 Usage

The value of semantic models for language proofs and verification is well established [ADKR86, Ame89b, Wol88]. Event-based operational models have the advantage of having a higher level of abstraction—indeed, it has been suggested that event histories may be the most natural abstraction of distributed systems [MH89]. The use of event-based systems for behavioural specification and analysis is wide-spread [Bat89, Bat87b, LL89, MH89] (see also chapter 2). Furthermore, they can be used as a technique for reasoning about the externally observable behaviour of objects—essential for testing the behavioural conformance and reusability of certain objects [VJN+90] without violating encapsulation. At the core of any such model is the fundamental set of event classes, the instances

[1] That which models externally observed behaviour rather that internal form or semantics.

of which describe the model's behaviour. Such an alphabet will be an important by-product of this chapter.

3.2.2 Choosing a Process Calculus

To facilitate the behavioural analysis of any system, an appropriate calculus must be used to maximize the efficacy of the study. In this case, the calculus should be capable of expressing observational behaviour independently of any detailed timing constraints. It should be event-based, to allow behavioural specification to variable levels of abstraction; the advantages of event-based models are well documented [Bat87b, Lar90, Bat87a, LL89, Smi85, Moh88]. It should provide inherent support for encapsulation and message passing—essential to object oriented systems. Finally, it should support all of these requirements *as simply as possible*. On the basis of all these criteria, the CSP [Hoa85] process calculus was chosen. Where essential, the Z notation [Spi89] is used to supplement CSP's power to express components of state. Note that we have not combined CSP and Z, we have merely used both to specify a complete model. CSP to describe the dynamic, process based elements of the model and Z to describe data types. These parts supplement each other, but there is no interaction between the formalisms and the separate consistency and coherence of both are preserved.

CSP aside, many other model substrates were considered, and rejected, on the basis of a short feasibility study made at the onset of the project. We considered the following calculi: *Petri-Nets* [Pet77, DGM88], which were abandoned due to their excessive genericity; *PROT nets* [BB88], which we considered to be too specific; *Finite State Machines*, which we rejected by virtue of the fact that they are unable to model all distributed systems [Bat87b]; *LOTOS* [Bri87], was discarded on account of its inordinate complexity; and the Calculus of Communicating Systems (CCS) [Mil89] which, is somewhat weaker than CSP for expressing process specifications [Bel89].

The choice of CSP, a process based calculus, as an analytical tool for modeling object oriented systems might seem counter-intuitive. Its rudimentary data-typing facilities would seem to offer poor support to a paradigm in which data is paramount (hence the use of Z). We assert that although CSP does suppress some aspects of object oriented systems, it emphasizes more important ones—those relevant to the externally observable behaviour of such systems within a parallel environment. By concentrating on this high level view of behaviour, we retain the abstractness of the model and thus its flexibility [Yel89].

It is assumed that the reader has a working knowledge of both CSP and Z. Although the list of symbols used is defined in Appendix B, we provide no introduction to the semantics of either formalism.

3.2.3 Requirements of Parallel Object Oriented Systems

The object oriented paradigm is used widely for a diverse range of applications ranging from databases to programming languages. Some have even proposed it as a general model for computing and yet its precise definition remains elusive [BGM89, Ren82]. A menagerie of techniques abound, all claiming to use an object oriented method or to be a vital part of an object oriented system, but no clear consensus exists on what constitutes such a system. There are many criteria sets for object oriented languages, some based on the mechanisms and syntactic constructs offered by languages [Weg90] and others based purely on the properties of these systems [BGM89]. We have consolidated these views to yield a *working consensus* of the required features of object oriented systems. These properties constitute the core of our model. They are:

- **Encapsulation**, through the use of objects;
- **Set-Based Abstraction**, through the use of prototypes or classes;
- **Behaviour Sharing**, through the use of inheritance or delegation; and
- **Operation Polymorphism**, through the use of message passing and receiver side binding.

In purely sequential environments, message passing is not essential. The use of strong typing means that all method invocation can be reduced to procedure calls. However, parallel systems necessitate message passing in order to avoid costly alternatives, for example shared memory (as distinct from shared address spaces which makes things simpler) [Ame89b].

In our model we endeavour to support all of these facilities, imbuing it with an operational conformity to a wide range of object oriented systems. In addition, it will support parallelism to method level using asynchronous message sending.

3.3 The Model

3.3.1 General Structure

Fundamentally, the model consists of an object space Σ and a *supervisor*—a process which initializes and maintains the space. The supervisor has two subordinate processes: the *generator* and the *scavenger*, which respectively oversee the creation and destruction of data objects within the space. This relationship is depicted in Figure 3.1. The supervisor exists, not as a top level of

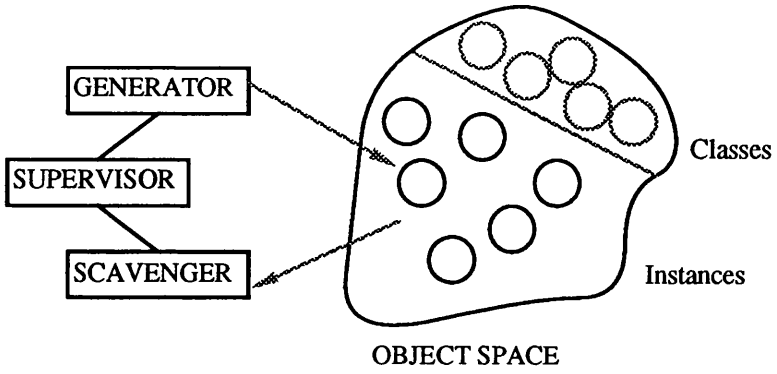

Figure 3.1: Model Overview

the hierarchy of control (as Meyer observes: object oriented architectures are decentralized and have no 'top' [Mey88]), but as a support service. It is a means of overseeing object allocation and deallocation. As such, it reflects a necessary entity in many parallel implementations to perform tasks such as load balancing or performance monitoring.

Σ contains a set of classes $C : \mathbb{P}\mathsf{CLASS}$ and a set of instances $I : \mathbb{P}\mathsf{INSTANCE}$, indeed $\Sigma = I \cup C$. Classes are templates for instance creation. A given class $c : \mathsf{CLASS}$ defines the subset of I, $I_c : \mathbb{P}\mathsf{INSTANCE}$, such that I_c contains only instances of c. Since each object is an instance of a class then:

$$I = \bigcup_{c \in C} I_c \tag{3.1}$$

The binary instantiation function $\sigma : \mathsf{INSTANCE} \rightarrow \mathsf{CLASS}$ links any instance $i : \mathsf{INSTANCE}$ with its parent class c:

$$\forall i . \sigma(i) = c \Leftrightarrow i \in I_c \quad \text{where} \quad i \in I \ \wedge \ c \in C \ \wedge \ I_c \subset I \tag{3.2}$$

Conversely $\sigma^{\sim}(c) = I_c$.

To avoid the conceptual problems of metaclasses (or other circularities), with an arguable loss of orthogonality, many object oriented models assert that classes are *not* objects [Mey88, CRS89]. Specifically, classes and objects may use an analogous message passing interface—but they are distinct entities related only by instantiation. We also adopt this policy, chiefly because we believe that classes are merely templates from which new instances are produced and because it simplifies the model. If, like objects, they are subject to change at run-time then system behaviour is more difficult to represent and manipulate. Indeed, within parallel systems we consider it is necessary to assume the invariant ΞC—the set C remains unchanged whilst the system is used [Ame89b]. This yields a system which is more tractable to formal verification.

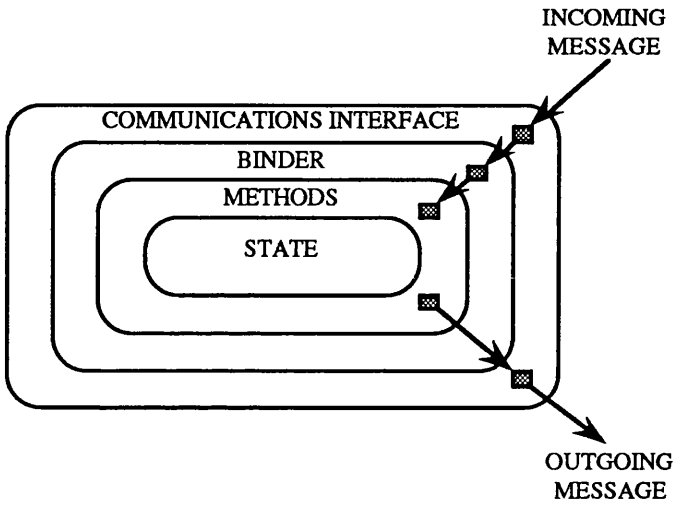

Figure 3.2: Instance Object Structure

Object instances are represented as CSP processes. Each encapsulates four separate parts:

- **a composite state**, which is visible only from inside the object;
- **a set of methods** which, when actuated, may alter or convey a facet of this state;
- **a binder**, which actuates methods depending on the messages it receives; and
- **a communications interface** which handles the posting and receipt of messages between objects (the sole means of communication).

This four layer architecture is depicted in Figure 3.2, each layer encapsulates those within. Objects request actions of each other by message passing. The receiver of a message responds by actuating one of its methods, according to a set of bindings defined by the receiver's class. Therefore, a message send is little more than a *request* for a certain internal behaviour. Incoming messages must pass through three layers of the object before they cause any activity in the host. Similarly, outgoing messages traverse two levels before they are broadcast.

All members of I are created on demand by the generator, in response to a request issued to members of the set C by other instances. Each instance responds to messages in a manner defined

by its class. All instances are mutually analogous in external form and behaviour, except for the set of messages to which they respond. Each may therefore be related to others by CSP's *change of symbol* operator. If the form of all instances can be generalized by the process P, then the minimal set of event classes which characterize object behaviour can easily be deduced, since the behaviour of P is uniquely specified by $(\alpha P, traces(P))$ [Hoa85].

Because our model focuses on the run-time properties of parallel, object oriented programs, the creation of new classes or the modification of existing classes is not addressed. Indeed, classes are not modeled as processes and their dynamic properties are not considered. They are considered to be static templates which respond to requests (notably the function '*new*') for instantiation by yielding new instances.

The parallel, object oriented run-time system is modeled as a parallel composition of a set of labeled *OBJECT* processes—an array[2] of communicating subsystems maintained by the *SUPERVISOR* process. Objects communicate using a communications bus, or *MESSAGE BUS* (defined in Section 3.3.3). The system may be expressed by composing, using remote subordination, this collection of objects with a communication system based on *MESSAGE BUS*, thus:

$$SYSTEM = (mb : MESSAGEBUS)//SUPERVISOR \tag{3.3}$$

where:

$$SUPERVISOR = (INIT \| (|||_{\mathcal{M} \geq n > 0}\, n : OBJECT))\hat{}(SHUTDOWN; SKIP) \tag{3.4}$$

Each *OBJECT* process executes concurrently and each has the same alphabet. However, because they are composed with ||| they are not *lock-step* synchronized. The message bus, a global resource for all objects, is the only other external process they jointly perceive.

INIT and *SHUTDOWN* are auxiliary processes responsible for the correct initialization and termination of the system. Their definition is system specific and tangential to our model. It is important to note that *INIT* must make appropriate use of the generator to create the minimum set of objects needed to bootstrap the system. Similarly, it is incumbent on the *SHUTDOWN* process to interrupt the recursive *OBJECT* processes at the appropriate time, to bring down the system gracefully by garbage collecting all objects out of existence. Because of the major consequences of *INIT* and *SHUTDOWN*, they may be guarded with an event such as *catastrophe* ($\lightning$).

[2]The object array is bounded by the theoretical maximum $\mathcal{M}$ in our model. This reflects a pragmatic limitation of resources.

3.3.2 Object Creation and Destruction

Objects are modeled by the process *OBJECT*, defined:

$$\begin{aligned} n : OBJECT = \; & \mu X.(n.st : STATE // \\ & (((n : ALLOCATE(\theta_n, j_n); j_n : RUNOBJ) \\ & \hat{}\, n : DEALLOCATE(j_n)); X)) \end{aligned} \tag{3.5}$$

This process set is a support service that allocates, maintains and deallocates objects to order. It is controlled by its environment. The event $n.allocate(\theta)$ causes the process $n : OBJECT$ to create and maintain a new object instance of class θ—providing that it is not already maintaining one. This instance may later be deallocated by the event $n.deallocate$, after which that instantiation of the *OBJECT* process is free to allocate and manage a new instance (since it is recursive). The function[3] $new(\theta)$ triggers this behaviour to yield fresh instances, by offering the event $[]_{\mathcal{M} \geq n > 0} n.allocate(\theta)$ to the environment via the *INIT* process. The use of non-deterministic choice ensures that only an *OBJECT* process that is ready to allocate an object is used and frees us from having to explicitly enumerate it. Each object has an encapsulated state modeled by the subordinate *STATE* process (as defined in Section 3.3.6).

Object allocation and deallocation are handled by the *ALLOCATE* and *DEALLOCATE* processes, which encapsulate functions governing the behaviour of the generator and scavenger processes respectively. $ALLOCATE(\theta, i)$ generates a new object of class θ and places a reference to it in i. For a trace, tr, it is specified:

$$ALLOCATE(\theta, i) \textbf{ sat } (\overline{tr}_0 = \checkmark \;\Rightarrow\; (\sigma(i^{\checkmark}) = \theta \;\wedge\; \theta \in C \;\wedge\; i^{\checkmark} \notin I \;\wedge\; \underline{i}^{\checkmark} \in I^{\checkmark})) \tag{3.6}$$

$DEALLOCATE(j)$ disposes of the reference j in an analogous way, freeing the memory thus occupied. It is similarly specified:

$$DEALLOCATE(j) \textbf{ sat } (\overline{tr}_0 = \checkmark \;\Rightarrow\; (\sigma(j) \in C \;\wedge\; j \in I \;\wedge\; j \notin I^{\checkmark})) \tag{3.7}$$

These processes are implemented thus:

$$ALLOCATE(\theta, i) = allocate(\theta) \rightarrow (i := \eta(\theta); I := I \cup i; (st.init!\emptyset \rightarrow SKIP)) \tag{3.8}$$

$$DEALLOCATE(i) = deallocate \rightarrow (st.dest!\emptyset; I := I \setminus i \rightarrow (\omega(i); SKIP))) \tag{3.9}$$

The events *allocate* and *deallocate* directly guard the activity of the generator and scavenger respectively. *Allocate* prefixes the generation of a new instance $i \in I_\theta$ (the actual generation of an

[3] Traditionally, *new* is a message understood by all class objects. However, as classes are not modeled as objects (see Section 3.2.3) or as processes (see Section 3.3.1) this representation is not applicable here.

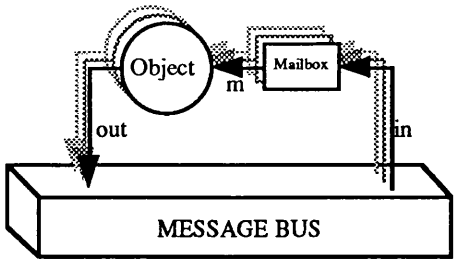

Figure 3.3: Object Communication Interface

instance is performed by the primitive generator function $\eta(\theta)$) and postfixes it with state initialization by communication on the channel *init* (see Section 3.3.6). *Deallocate* guards the success ($\langle\surd\rangle$) and subsequent termination of *STATE*, and the use of the scavenger to reclaim memory (achieved by the primitive scavenger function ω). It interrupts all activity within an object—an essential precaution to ensure safe deallocation. An object *cannot* refuse to be deallocated since (because of the ^ operator):

$$\forall s : s \in traces(OBJECT) \;.\; deallocate \notin ref(OBJECT/s) \tag{3.10}$$

Once spawned, each object instance is handled by its own *RUNOBJ* process which scans incoming message traffic and reacts to it. It is defined:

$$RUNOBJ = MAILBOX \;||\; (BINDER \;||\; FRESULT \;||\; HERROR) \tag{3.11}$$

Here, the *MAILBOX* process (defined in Section 3.3.3) splits incoming message traffic according to its type. Process *BINDER* accepts execution requests and schedules the appropriate method, process *FRESULT* forwards incoming execution results to the appropriate *METHOD* process and *HERROR* ensures that error notifications are handled correctly. All are fully defined in Section 3.3.5.

3.3.3 Object Exterior: The Communications Interface

All inter-object message traffic is handled by a *message bus* to which they are all connected via the private channels *in* and *out*. Each object broadcasts messages directly to the bus (via *out*) and receives them from its mailbox buffer, which in turn reads them from *in* as depicted in Figure 3.3.

The message bus is a set of processes that maintain object communications and route all messages to their destination mailboxes. It may be thought of as a finite number[4] of concealed

[4]The number of channels is bounded by the theoretical maximum C.

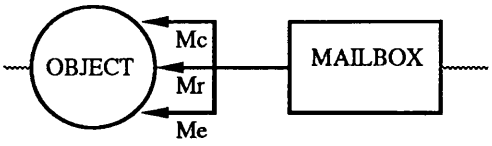

Figure 3.4: The Mailbox Buffer's Three Channel Split

relay channels, which may be non-deterministically allocated to facilitate a temporary link between one object's *out* channel and another's *in*. The alternative strategy, maximal interconnection, has a channel complexity $O_c(n) = n(n-1)$. For the message bus, however, it is only of order n. This technique also has the advantage of circumnavigating the need for dynamic channel allocation, which is prohibited by recent versions of CSP [Hoa85]. The message bus is a global resource, shared by object processes using remote subordination. A detailed analysis of the message bus implementation is not germane to this work, at a rudimentary level it may be specified:

$$MESSAGEBUS = (\underset{C \geq y > 0}{\|} \mu X(y.out?m \rightarrow ((m.dest).in!m \rightarrow X))) \tag{3.12}$$

Where m is an arbitrary message and $m.dest$ is the message destination which, as we will see later, is one of the fields of the message packet itself. C is the maximum number of concurrently available channels within the model. Theoretically, C may have any positive value, although systems with a C of unity would not benefit from message passing parallelism. Each object is locked into cyclic interaction with the message bus. Within an object, the bus is referred to by the label *mb* and is used by a statement conforming to:

$$\ldots(\underset{C \geq y > 0}{[]} mb.y.out!x \rightarrow \ldots)\ldots \tag{3.13}$$

Here x is the message to be broadcast and y is the non-deterministically selected, message bus channel which conveys this message.

All messages consist of packets, the contents of which are explained later in this section.

$$\alpha in(OBJECT) \equiv \alpha out(OBJECT) \equiv \alpha m(OBJECT) \equiv \mathsf{PACKET} \tag{3.14}$$

The mailbox buffer is an essential aspect of the model's support of asynchronous message sending and it prevents sporadic bursts of incoming message traffic from unduly loading OBJECT processes. Because of the requirement for asynchrony, the model abandons the usual remote procedural call (RPC) protocol. As a consequence, some method of differentiating between incoming messages and incoming results from messages already broadcast is needed. Our solution is to provide each message with a *type*. All messages is one of the following three types:

- **Conventional Messages**, which are legitimate requests for method execution;
- **Return Messages**, which contain results in response to those above; and
- **Error Messages**, which signal the failure of the binder or an actuated method.

The mailbox splits all incoming messages, on the channel *in*, according to type and outputs them on three channels: M_c, M_r and M_e (represented as the compound channel m in Figure 3.3), as shown in Figure 3.4. The *MAILBOX* process is defined by the equations:

$$MAILBOX = (BUFFER \gg M)^\wedge(error(x) \rightarrow STOP) \tag{3.15}$$

where:

$$BUFFER_{\langle\rangle} = left?x \rightarrow BUFFER_{\langle x\rangle} \tag{3.16}$$

$$BUFFER_{s^\wedge\langle x\rangle} = left?y \rightarrow BUFFER_{\langle y\rangle^\wedge s^\wedge\langle x\rangle} \,|\, right!x \rightarrow BUFFER_s \tag{3.17}$$

and:

$$\begin{aligned} M = \quad & \mu X.(left?x \rightarrow ((M_c!x \rightarrow X) \not\leftarrow (x.type = conventional) \\ & \not\rightarrow ((M_r!x \rightarrow X) \not\leftarrow (x.type = return) \not\rightarrow (M_e!x \rightarrow X)))) \end{aligned} \tag{3.18}$$

where $x.type$ is the *type* of message x.

Evidently for the process $i : OBJECT$, messages are broadcast on $i.out$ and received on M_c, M_r and M_e via $i.in$, thus:

$$\alpha i.out = \{x | c!x \in \alpha(i : OBJECT)\} \tag{3.19}$$

$$\alpha i.in = \{x | c?x \in \alpha(i : Mailbox)\} \equiv \alpha i.M_c \cup \alpha i.M_r \cup \alpha i.M_e \tag{3.20}$$

All objects are instantiated with channels M_c, M_r, M_e and $i.in$. The channel $i.out$ is virtual and triggers message bus activity. Consequently, all channel allocation is static. The concept of virtual channel and non-deterministic channel selection can be used in any application where some form of proof is required, but static channeling is overly restrictive.

Each message communicated on these channels is a 6–tuple (of type PACKET), defined:

$$\mathsf{PACKET} == (\mathsf{SRC} \times \mathsf{DST} \times \mathsf{TYP} \times \mathsf{C} \times \mathsf{CAT} \times \mathsf{TIME}) \tag{3.21}$$

Wherein each subcomponent is defined:

- SRC: **Source**, the identity of the object, method, thread and future channel, if appropriate (see later), from which the message was originally broadcast;

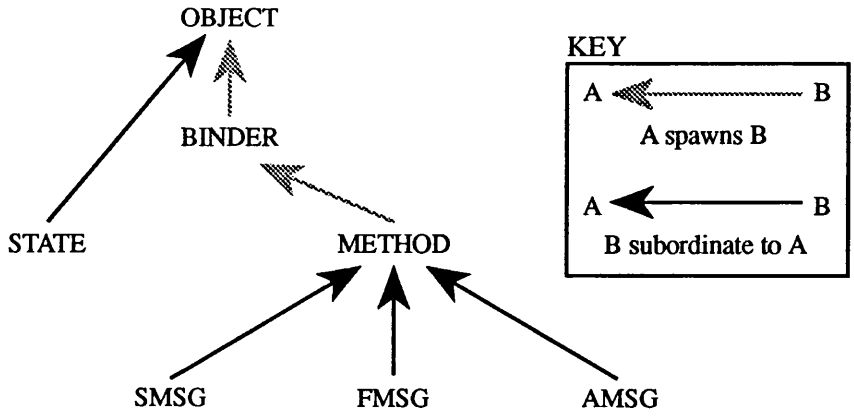

Figure 3.5: Object Subordination Hierarchy

- DST: **Destination**, the identity of the target object;
- TYP: **Type**, the type of the message, as defined above (see page 76);
- C: **Contents**, the contents of this field are type dependent. Conventional messages contain a message selector and zero or more arguments, return messages contain a result and error messages contain an error notifier;
- CAT: **Category**, in a parallel object oriented system, where invoked methods may be run concurrently with their callers, it is often necessary to distinguish between three types of method scheduling behaviour:
  - **Synchronous Messages**, which block the caller until the callee returns a result [GR83].
  - **Asynchronous Messages**, which allow the caller to continue immediately and ignores any generated result [Car89].
  - **Future Messages**, which allow the caller to continue until the result is required. When a result is required, if the callee has not finished the caller is blocked awaiting it, otherwise the result is immediately accessible [HW89]; and
- TIME: **Timestamp**, the time the original conventional message was broadcast. This acts as a unique identifier for each message and is a means of ordering them. This poses difficulty in a distributed systems where meaningful timestamps cannot be achieved without the use of a logical (Lamport) clock [Lam78, LMS85].

All message sending activity is initiated during the execution of method codebodies (modeled by instances of the *METHOD* process, defined in Section 3.3.5). Depending on the category of

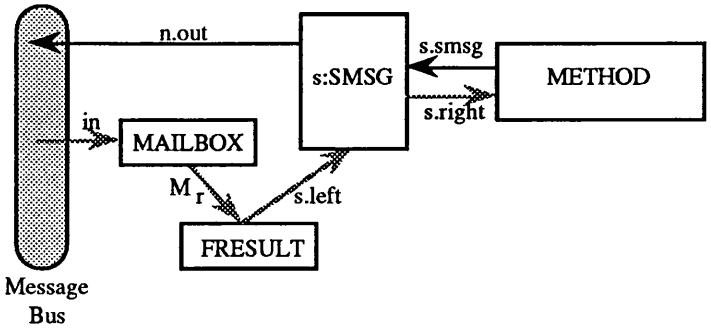

Figure 3.6: A Method's View of a Synchronous Communication

message scheduling desired, *METHOD* processes may invoke one of three subordinate processes: *SMSG*, *FMSG* and *AMSG* to cope with synchronous, future and asynchronous communications respectively. The relationship between these processes is depicted in the subordination hierarchy of Figure 3.5. Each of these processes broadcasts an outgoing packet on *i.out*, but only *SMSG* and *FMSG* await a result on channel M_r. In this case, when a result arrives, it is returned to the scope of the sending subordinate through the process *FRESULT*'s channel *left* (see Section 3.3.5) and to the calling method via channel *right*—as depicted in Figure 3.6. The subordinate $s : SMSG$ is defined[5]:

$$s : SMSG = smsg?packet_s \rightarrow (\mathop{[]}_{c \geq n > 0} mb.n.out!packet_s \rightarrow (left?r_s \rightarrow (right!r_s \rightarrow SMSG))) \tag{3.22}$$

and used:

$$sync(packet) \equiv s.smsg!packet \rightarrow (s.right?result \rightarrow \ldots) \; \backsim \tag{3.23}$$

so that the method and its subordinate *SMSG* are blocked whilst awaiting a result on $s.left$.

Future message scheduling is handled in an analogous manner. However since futures do not block initially, a number of outstanding futures may exist at any given time (to permit only one is to invite deadlock). Consequently each *METHOD* process needs a finite collection[6] of future message handlers in its $f : FMSG$ subordinate:

$$\begin{aligned} f : FMSG = \; & \mathop{\|}_{\mathcal{F} \geq x > 0} fmsg.x?packet_{fx} \rightarrow \\ & (r_{fx} := \emptyset; \mathop{[]}_{c \geq q > 0} mb.q.out!packet_{fx}(source.future = x) \rightarrow \\ & (left.x?r_{fx} \rightarrow (right.x!r_{fx} \rightarrow FMSG))) \end{aligned} \tag{3.24}$$

FMSG processes are used in two phases: *fsend* and *frec*, defined:

$$fsend(packet, x) \equiv \mathop{[]}_{\mathcal{F} \geq x > 0} f.fmsg.x!packet \quad frec(x) \equiv f.right.x?result \tag{3.25}$$

[5] In all these equations, $\emptyset$ represents a nil value appropriate to the type concerned.
[6] The number of outstanding futures is bounded by the theoretical maximum $\mathcal{F}$.

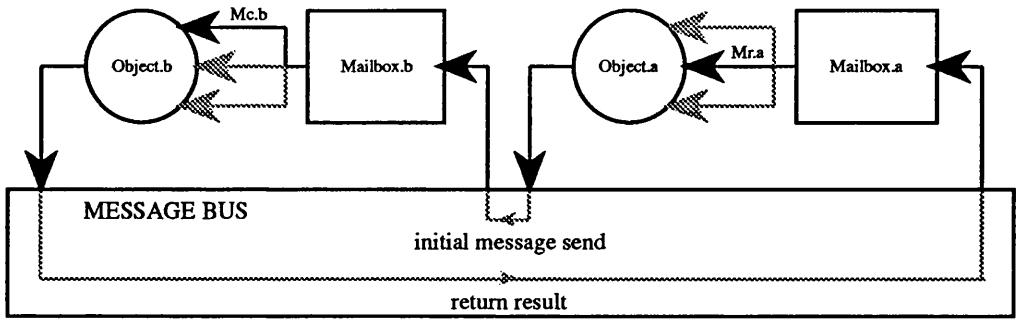

Figure 3.7: The System's view of a Synchronous Communication

A *FMSG* subordinate defines a group of servers, each *fsend* use allocates one server (non-deterministically) and sends the message. On execution, *fsend* instantiates x—the future channel identifier, which uniquely identifies the waiting future. *Frec* must guard all usage of the result thereafter. Between the usage of *fsend* and *frec*, *FMSG* server x is blocked awaiting the result—but the client *METHOD* may continue execution *immediately*. If the result is returned before *frec* is used, *METHOD* will experience no delay when *frec* is finally used to read the result, otherwise when *frec* is used *METHOD* will block. This action models the protocol of future messages. Note that the future channel identifier, x, forms part of the packet of outgoing future communications (it is part of the source field), hence the parameterization of *mb.q.out!* in Equation 3.24. Consequently, return messages know which future channel to be directed to via *FRESULT* (see Figure 3.6 and Section 3.3.5).

Asynchronous message scheduling is the simplest as no return is expected. The packet is forwarded and the sender terminates immediately. It is defined as a subordinate $a : AMSG$:

$$a : AMSG = amsg?packet_a \rightarrow (\mathop{[]}_{C \geq n > 0} mb.n.out!packet_a \rightarrow AMSG) \tag{3.26}$$

and used:

$$async(packet) \equiv a.amsg!packet \tag{3.27}$$

Once launched, a message is routed, by the message bus, to the mailbox of the object to which the packet field *destination* refers. Consequently the sending object, x (with name space $\mathcal{N}_x$, see Section 3.3.6), must have a reference to the target object. More formally[7], for an arbitary object x, executing a method process y, involving packet p and future channel z, we assert that:

$$\mathfrak{D}(sync(p)) \equiv \mathfrak{D}(fsend(p,z)) \equiv \mathfrak{D}(async(p)) = (p.destination \in (\mathbf{ran}\ \mathcal{S}_x \cup \mathbf{ran}\ \mathcal{S}_{xy})) \tag{3.28}$$

[7] $\mathfrak{D}$ is the CSP *domain* operator.

Conventional messages may cause the destination object to spawn new *METHOD* processes and return their results, if any. Alternatively, it may return an error message if the message selector is outside the domain of selectors it understands (D_M, see Section 3.3.6) or subsequent execution of the bound method generates an error condition. Result and error messages are forwarded to the process that requires them asynchronously. The send and receive cycle for a synchronous (or future) conventional message send (from an object a to an object b) is depicted pictorially in Figure 3.7.

3.3.4 Inheritance

Inheritance is primarily a composition mechanism [Coo86, Ame89b, Cus89], which allows classes to be partially ordered according to the relations of inclusion between their properties. Thus, as each instance of an object has its own instance variables modeled on those of its class, so subclasses of a class have their own method protocol which includes that of their superclasses. There are two principal methods of achieving this within our model:

- **Additive Inheritance.** Each classes' method dictionary, D_M (see Section 3.3.6), contains copies of the entries existing in their superclass's dictionary, in addition to their own. Method lookup need thus consult one dictionary only: the local one, and data encapsulation is not violated. Furthermore the implementation of such a system does not require shared memory. The POOL model is based on this concept [Ame87].
- **Delegative Inheritance.** Each class has its own dictionary and a reference to its superclass, to which instance method binding lookup may pass if it has failed locally. In this way, the search continues until the *root* superclass is reached. This method is very flexible, especially if the ΞC invariant (see Section 3.3.1) is relaxed. Furthermore, it may be used to represent delegation. The Smalltalk and SOLVE models are based on this concept [GR83, RWW88b].

Both of these techniques are essentially representations of implementation inheritance [BGM89, PW91a]; specification inheritance is widely condoned, but no consensus yet exists of its most apt utility and form. We use the delegative inheritance format primarily because of its flexibility. This can be modeled using a class delegate sequence (CDS) process which is a subordinate to the *BINDER* process defined in Section 3.3.5. A CDS is a sequence, defined:

$$CDS = cds : SEQ_{it} \tag{3.29}$$

where it is the inheritance trace, a sequence of classes of length m, such that it_0 is the instance's own class and:

$$\forall i,j \ : \ 0 \leq i,j < m \ . \ (it[i] \sqsubseteq it[j] \ \Rightarrow i \leq j) \tag{3.30}$$

Here '$\sqsubseteq$' is a reference to any partial ordering relation suitable for behavioural inheritance, e.g. Cusack's relation [Cus89]. For single inheritance, the sequence represents a successive series of superclasses dictating the order of method lookup. For multiple inheritance it is merely the normalized[8] list of superclasses from an acyclic graph. For systems using delegation it simply represents a delegation sequence (or graph).

Other approaches to modeling inheritance in CSP (e.g. [Cus89]) graft the concept of inheritance onto the CSP language itself. We did not use this approach for two reasons. Firstly, such a graft would fix the semantics of inheritance and hide its explicit use by making inheritance lookup a primitive of CSP. Secondly, by altering CSP in this manner, one may invalidate the rules that bind various aspects of the language—thus compromising the whole. Without a formal review, the ramifications of such alterations are unknown.

3.3.5 Intra-Object Behaviour

Recall how we described, at the end of Section 3.3.2, how an *RUNOBJ* process splits messages into three channels and responds differently to messages on each channel. In fact:

- Incoming message selectors on the channel M_c are bound to the relevant method (which is scheduled) by the *BINDER* process;
- Incoming results on the channel M_r are forwarded to the appropriate subordinate by the process *FRESULT*; and
- Incoming notifiers on the channel M_e are handled by the *HERROR* process.

These processes are further defined:

$$\begin{aligned} b : BINDER = \ & \mu X.(M_c?m_b \rightarrow \\ & (BIND(m.content, 0)|||X))\hat{}(error(cds) \rightarrow STOP) \end{aligned} \tag{3.31}$$

$$\begin{aligned} f : FRESULT = \ & \mu X.(M_r?m_f \rightarrow ((\textbf{if } \mathfrak{D}m_f.source \textbf{ then } left.(m_f.source)!m_f) \rightarrow X) \\ & \hat{}(error(x) \rightarrow STOP)) \end{aligned} \tag{3.32}$$

[8] Such normalization is a normal part of conflict removal in multiple inheritance systems, wherein the linearized path obtained by DAG normalization is used to determine the order of priority of co-named methods inherited by an instance.

$$HERROR = \begin{array}{l}(M_e?x \rightarrow (error(x.content) \rightarrow NOTIFY(x.content))) \\ |(error(x) \rightarrow NOTIFY(x))\end{array} \tag{3.33}$$

BINDER accepts input from the channel M_c and spawns a *BIND* process to initiate the method signified by the selector in *m.content*. The *BIND* process searches the local and non-local method dictionaries for the selector. If it is found, a *METHOD* process is spawned to execute the codebody associated with it and the *BIND* process terminates. By this time, *BINDER* has already spawned another.

$$BIND(x,n) = \begin{array}{l}lookup(x,n) \rightarrow \\ ((s : SMSG//f : FMSG//a : AMSG//METHOD(x, cds.it[n])) \\ \quad \nleq x \in \mathbf{dom}\ D_{M_{cds.it[n]}} \ngtr BIND(x, n+1)) \\ \nleq \mathfrak{D} cds.it[n] \ngtr (error(cds) \rightarrow STOP)\end{array} \tag{3.34}$$

Here, the event $lookup(x,n)$ denotes the search in $D_{M_{it[n]}}$ for message identifer $\tilde{x}$. Because of the interleaving used in Equation 3.31 it is possible for more than one *METHOD* process to execute at once. Starting with the class of the receiver, $cds.it[0]$, *BIND* progressively probes deeper into *cds* until a match is found. If the bounds of *cds* is exceeded before a match occurs, the event $cds.error(cds)$ will occur, interrupting *BINDER* and triggering *HERROR*. The *METHOD* process extracts the relevant codebody from the class defining the method, executes it and returns a result packet r_m (inheriting the source, category and timestamp of its originator m), which is forwarded to the mailbox of the object originating the communication (providing the originating message was not asynchronous):

$$METHOD(m,\theta) = \begin{array}{l}execute(m,\theta,r_m) \rightarrow ((async(r_m) \rightarrow SKIP) \nleq \\ (m.category \neq async) \ngtr SKIP)\end{array} \tag{3.35}$$

As a result of the use of function $new(\theta)$ (see page 74) by executing codebodies, new variables may be effectively introduced into *METHOD* processes. These represent bindings (of names to object identifiers) local to the *METHOD* process and outside of $\mathcal{S}_x$ (see Section 3.3.6). For a process $y : METHOD$, spawned from $x : OBJECT$, these extra mappings are denoted $\mathcal{S}_{xy}$. Note that:

$$\mathbf{ran}\ \mathcal{S}_{xy} \subset \mathbf{ran}\ \mathcal{S} \tag{3.36}$$

and:

$$acc(y : METHOD) \setminus \{m, \theta, r_m\} = \mathbf{dom}\ \mathcal{S}_{xy} = \mathcal{N}_{xy} \tag{3.37}$$

See also Equation 3.28.

FRESULT accepts input from M_r. It extracts the *source* field information from the incoming result and forwards the result to the channel associated with it (see Figure 3.7). In the way prescribed by equation 3.32, *FRESULT* returns results to the thread (and where necessary the

future channel) indicated only if this thread still exists. Like *BINDER*, *FRESULT* is halted by the occurrence of an $error(x)$ event.

HERROR accepts input from the channel M_e, in response to which it generates an $error(x)$ event which eventually halts both *BINDER* and *FRESULT*. The process reacts identically if another process generates the $error(x)$ event (e.g. *BINDER*). The handler then informs the user that the system has failed using the *NOTIFY* process, which is parameterized by the fault cause. For example, $NOTIFY(cds)$ might inform the user that a failure occurred because an incoming selector could not be bound correctly. *NOTIFY* is not further defined here, except that it should end in *STOP* so that the 'death' of the faulty *OBJECT* is complete. This termination response seems pessimistic, but often it is the most prudent course of action [KS90], especially in a heavily parallel system in which recovery might jeopardize the consistency of processes dependent on the failing one.

3.3.6 Object Interior: The Composite State

An object's potential behaviour depends entirely on two interdependent aspects of its construction: the static parts of its structure which depend on the class of which it is an instance, e.g. its method and instance variable type dictionaries D_M and D_I; and the dynamic constituents, the contents of these variables at run-time—i.e. *state*. Assuming the data type definitions [Spi89]:

$$\begin{array}{ll} \mathsf{NAME} == \mathrm{seq}_1\ \mathsf{CHAR} & \mathsf{ARGTYPE} == \mathrm{seq}\ \mathsf{CLASS} \\ \mathsf{RESULTTYPE} == \mathsf{CLASS} & \mathsf{INSTANCE} == (\mathsf{STATE} \times \mathsf{CLASS}) \\ \mathsf{METHODTYPE} == (\mathsf{ARGTYPE} \times \mathsf{RESULTTYPE}) & \end{array} \quad (3.38)$$

Where CHAR represents the alphanumeric character type and ARGTYPE and RESULTTYPE are the types of the arguments for conventional and return message sends. Then D_M, D_I, and STATE can be represented as the mappings:

$$\begin{array}{ll} D_M : \mathrm{seq}\ ((\mathsf{NAME} \times \mathsf{METHODTYPE}) \mapsto \mathsf{CODEBODY}) & D_I : \mathrm{seq}\ (\mathsf{NAME} \mapsto \mathsf{CLASS}) \\ \mathsf{STATE} : \mathrm{seq}\ (\mathsf{NAME} \mapsto \mathsf{OBJID}) & \end{array} \quad (3.39)$$

Here, OBJIDs are object identifiers or references. Note that the cartesian domain of D_M supports method polymorphism and overloading by increasing the significance of type over naming conventions. Each class, θ, defines its own dictionaries $D_{M\theta}$ and $D_{I\theta}$. Indeed, as classes are not represented as processes, they are completely defined by their name and dictionaries.

It is important here to distinguish between the function STATE and the process *STATE*. STATE is a graph of NAMEs and INSTANCEs, a function which maps identifiers onto the INSTANCEs they represent, thus defining an object binding environment. To avoid confusion we shall in future refer

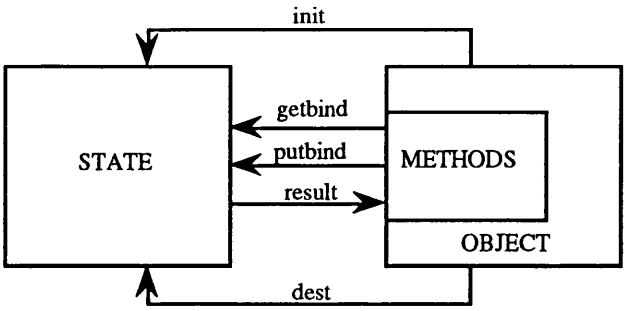

Figure 3.8: STATE-METHOD interprocess communication

to it as S and note that $I = \mathbf{ran}\ \mathcal{S}$. *STATE* is the CSP process that supports the abstraction S. STATE is a partial function, its domain, N, is called the *name space* of an object.

Within each object process, $x : OBJECT$, the subordinate process $x.st : STATE$ supports and encapsulates an object's state $\mathcal{S}_x$—that subset of S visible from object x. Note that:

$$\mathcal{S} = \bigcup_{i \in I} \mathcal{S}_i \quad \text{and} \quad \mathcal{N}_x = \mathbf{dom}\ \mathcal{S}_x \tag{3.40}$$

STATE is responsible for:

- At the outset, allocation and initialization of object state;
- At any subsequent time, state selection and alteration—the yielding and forging of bindings; and
- Finally, deallocation and destruction of the object.

Allocation and deallocation is performed by the *ALLOCATE* and *DEALLOCATE* processes (as defined in Section 3.3.2).

The function *id* maps (surjectively[9]) identifier names onto instance references, given a particular state:

$$id : (\mathsf{STATE} \times \mathsf{NAME}) \twoheadrightarrow \mathsf{OBJID} \tag{3.41}$$

This is an essential part of state selection.

The *STATE* process is controlled from the process *OBJECT* and from spawned *METHOD* processes, though five channels. These are pictorially represented in Figure 3.8.

[9]This function is a partial surjection because its range is complete whereas their domain is not [Spi89].

It is defined for an object x of class θ:

$$
\begin{aligned}
STATE = \quad & (init?z \rightarrow (\mathcal{S}_x := \{\}; \\
& \quad \textbf{foreach}\ ((\xi \mapsto x) : D_{I\theta}).\{ALLOCATE(x, j);\ \mathcal{S}_x := \mathcal{S}_x \oplus (\xi \mapsto j)\} \\
& \quad \rightarrow STATE)) \\
| \quad & (dest?z \rightarrow (\textbf{foreach}\ ((\xi \mapsto j) : \mathcal{S}_x).\{\mathcal{S}_x := \mathcal{S}_x \setminus (\xi \mapsto j);\ DEALLOCATE(j)\} \\
& \quad \rightarrow SKIP)) \\
| \quad & (getbind?\xi \rightarrow \\
& \quad ((result!id(\mathcal{S}_x, \xi) \rightarrow STATE) \not\in (name \in \mathcal{N}_x) \not\ni \\
& \quad (result!\emptyset \rightarrow STATE)) \\
| \quad & (putbind?\xi \rightarrow (putbind?j \rightarrow \\
& \quad (\mathcal{S}_x := \mathcal{S}_x \oplus (\xi \mapsto j) \rightarrow STATE)))
\end{aligned}
\tag{3.42}
$$

Here the operator **foreach** $((x : xs).\{\mathsf{action}(x)\})$ is an iterator, which performs the specified action (parameterized in x) for each item x in the sequence xs, such that:

$$
\textbf{foreach}\ (x : \langle x_0, x_1, \ldots, x_n \rangle).\{A(x)\} \equiv \begin{array}{l} A(x_0); A(x_1); \ldots; A(x_n) \\ \text{where } \forall x\ .\ \checkmark \in \alpha A(x) \end{array} \tag{3.43}
$$

Note that, in Equation 3.42, object state allocation is guarded by the event *init*?. When triggered, all the instance variables pertinent to the instance (as defined by $D_{I\theta}$) are allocated and added to the state $\mathcal{S}_x$. Event *dest*? guards the converse process. The event *getbind*? guards a name lookup, the name ξ is supplied as an argument and the object reference with this name (if it exists) is returned on channel *result*. Event *putbind*? guards the assignment of a new binding j to name ξ.

3.3.7 Data Encapsulation

Data encapsulation between objects is modeled very naturally on the inherent process encapsulation of CSP. Notwithstanding the ability of a process to subordinate others, or to communicate information to others via the prescribed channels, concurrent composition demands that for $(P||Q)$ the set of variables accessible by process P may not include any of these writable by process Q and *vice versa* [Hoa85]. Formally:

$$
acc(P) \cap var(Q) = var(P) \cap acc(Q) = \{\} \tag{3.44}
$$

and

$$
var(P) \subseteq acc(Q) \tag{3.45}
$$

State encapsulation is achieved by having each object's visibility limited to the name space of its subordinate *STATE* process (see Section 3.3.6) and any object names passed to it by message passing. However, concurrently composed methods of the same object, which share access to that

Parent Process	*CSP Event*	*Semantics*
MESSAGE BUS	$[]_{n>0} mb.n.out!$	message broadcast
STATE	$init!$	state initialization
STATE	$dest!$	state destruction
STATE	$getbind!?$	state selection
STATE	$putbind!!$	state alteration
STATE	$allocate(x,i)$	object allocation
STATE	$deallocate(i)$	object deallocation
METHOD	$smsg!$, $fmsg!$, $amsg!$	sync, future and async communication
METHOD	$f.right.x?$	future synchronization
METHOD	$execute(m,\theta,r)$	method executes
METHOD	$\langle\checkmark\rangle$	method terminates
BIND	$lookup(x,n)$	inheritance lookup
BIND	$error(cds)$	error during inheritance lookup
FRESULT	$M_r?$	object receives a result
BINDER	$M_c?$	object receives communication
HERROR	$M_e?$	object receives error notification

Table 3.1: Fundamental Event List

object's state via CSP's shared resource by interleaving model, obviously cannot be protected from mutual interference by these means. Serialization of object state access may be provided through the use of semaphores, monitors, or an acquire-release mechanism [Hoa85].

3.4 Fundamental Alphabet

From a textual examination of the CSP rules of our model, a fundamental alphabet of object behaviour may be built. The use of such model alphabets is discussed in Sections 3.2.1 and 3.5. An event alphabet is built by traversing the CSP description, noting each event that occurs *conditionally* from those that preceded it. This criteria removes redundant events, for example the event b in $a \rightarrow (b \rightarrow SKIP)$, but not in $(a \rightarrow (b \rightarrow SKIP)|(c \rightarrow SKIP))$. Using this method, the salient events of the model can be isolated. They are listed in Table 3.1.

There is still some redundancy in the event list of Table 3.1 because it considers each CSP rule in isolation. Due to the relationship between various rules and the architecture of the model, some events can be pruned from this table to produce a refined list. For example, the event $f.right.x?$ is part of the $frec(n)$ future synchronization sequence; in practice it is always followed by a variable binding event $putbind!!$ and is thus redundant. In an analogous manner $M_c?$, $M_r?$ and $M_e?$ are obviated by *METHOD* communications (on $smsg$, $fmsg$ and $amsg$), $\langle\checkmark\rangle_{METHOD}$ and $error(x)$ respectively. We further assume that state initialization and destruction are always accompanied

CSP Event	*Semantics*	*Parameters*
init!	Object creation	object class, reference to new instance
dest!	Object destruction	object class, object reference
bind	State access	object reference, read/write access
msg!	Message send	message selector, arguments, source, destination, category, timestamp
execute	Method execution	method name, class, host instance
$\checkmark_{METHOD}$	Method termination	method name, class, host instance, result
lookup	Inheritance or delegation lookup	object class, superclass
error	System Error	object class, instance reference, reason

Table 3.2: Refined Fundamental Event List

by the relevant memory management, therefore $allocate(x, i)$ and $deallocate(i)$ are not required. All of these omissions simplify the resulting alphabet and raise its level of abstraction, without compromising its expressive power.

Further refinement is possible since many events are very similar and warrant a single, parameterized event in their stead. For example, the events $[]_{c \geq n > 0} mb.n.out!$, *smsg*!, *fmsg*! and *amsg*! all represent the sending of messages of various categories and are better grouped into the parameterized event *msg*!. Also, *getbind*!? and *putbind*!! signify read and write access to object state and are best unified as *bind*. This simplification and refinement is only possible because of the formal nature of our model. Event alphabets are by no means unique (alphabets are presented in [BTM89, CW89]), but no other alphabet we have seen has been customized for an object oriented system and formally examined and refined in such a manner.

After this refinement, the fundamental events of Table 3.2 remain. The eight events, parameterized as shown, uniquely define the behaviour of the model. No two different behaviour patterns map to the same event sequence: the alphabet is behaviourally *injective* with respect to the model. By implication, since our model exhibits all of the common features of object oriented systems, this alphabet applies to them.

The existence of the *create object* event may seem redundant as, in many systems, it may be represented by the partial specialization of the *method termination* event[10]:

$$CREATE(class, obj) \equiv TERMINATE(\text{'new'}, class, \star, obj) \quad (3.46)$$

Where the result of the method termination, obj, is the new instance. We distinguish between them because many languages permit the static allocation of class instances via means other than the class method *new* [RWW88b, Str86, Cox86].

[10] $\star$ is used here as a wildcard instance.

3.5 Applications of this Model

The model has yielded a number of useful facts about the implementation of parallel object oriented systems. For example, it demonstrates the facility of allocating messages a type, as shown in Section 3.3.3, if the architecture of a system cannot directly support remote procedural call (see also [YT87, AH87, Agh90]). This system also provides a unified framework for the use of exceptions, signals and assertion failures through the message type *error*. Furthermore, the model demonstrates the need for effective deployment of buffers to speed up system response. Most significantly, it establishes that, from an operational perspective, object behaviour may be expressed as a sequence of parameterized events of just eight classes.

Once equipped with the complete alphabet of a system and a suitable language to facilitate the creation of powerful abstractions over the primitive events, any behaviour of the system may be expressed. Such expression may be used to:

- monitor and convey actual behaviour;
- define desired behaviour; or
- create behavioural mappings between otherwise non-isomorphic behaviours.

This thesis concerns only the former two applications.

An object oriented system may be instrumented to generate the events we have defined in order to relay details of its behaviour to an external monitor. This would allow tools such as debuggers, or performance analyzers, to present a textual display of program behaviour at a higher level of abstraction than program statements [Cox88a]. Perhaps more usefully, event streams may be mapped onto graphical representations of system behaviour, facilitating algorithm animation (e.g., [BH90b, KG88]). The completeness of the alphabet ensures the continued consistency between system state and its visual representation. Currently, visual event monitoring systems use small, unproven alphabets which restricts their utility and rigour.

In addition to providing a framework by which debuggers and other tools might monitor actual behaviour, their functionality might further be enhanced by allowing them to *specify* desired behaviour using this alphabet. Differences between actual and desired behaviour can also be expressed. [PW89] uses this work to design a debugger for a parallel object oriented system which has a full alphabet. Specifications may also be used as a means of error handling in parallel object oriented systems [PW90].

Event-based behavioural definitions based on this alphabet might be used to circumnavigate protocol incompatibilities between objects—facilitating reuse where otherwise it would have been impossible [VJN+90]. Event sequences of the client object are mapped to those of the server and vice versa, thus overcoming the different protocols without altering the objects themselves. Here the behaviour descriptions are used as 'glue' between the objects. The completeness of the alphabet is essential to the power of this glue.

3.6 Limitations

Currently, our model does not directly address issues such as garbage collection, the handling of momentous systemwide events ($\lightning$), the implementation of channel allocation techniques and state access serialization. All of these issues are significant problems in real systems and need to be more thoroughly examined. The advent of multiple inheritance and conformance throws the weaknesses of the CDS into sharp relief. A more abstract model of behaviour sharing is required, ideally one which supports specification inheritance in addition to code sharing. The recent popularity of language support for reliability [Mey89, KS90] indicates the need for the integration of preconditions, postconditions and exception handlers into our model.

Our model makes many simplifying assumptions which counteract its generality. Firstly, in offering no support for metaclasses we have limited its ability to model the set of systems based on languages which support this singularity (e.g. Concurrent Smalltalk [YT87]). However, we cannot *informally* conceive of this having any effect on the resultant alphabet. The model also assumes that environments with multiple inheritance linearize their hierarchy (see Section 3.3.4), before lookup, to detect clashes (e.g. as in Solve, Concurrent Smalltalk [RWW88b, YT87]), but there are environments without this common behaviour (e.g. Orient64/K [TI86a, TI86b]).

3.7 Conclusions

This chapter shows how we have constructed a model of a simple parallel, object oriented system using the formalisms of CSP and Z. In addition to demonstrating how these calculi may used together, we have demonstrated how the need for dynamic channel allocation can be circumnavigated in CSP. Analysis of our model has allowed us to enumerate the fundamental alphabet of object behaviour and the need for message typing. We find that *all* object behaviour may be represented as a partial ordering of events of only eight types.

The fundamental alphabet of event classes, produced as a by-product of this model, suggests another avenue of research in itself. Specifically, it prompts the development of a calculus which is capable of expressing object behaviour in terms of these events; and the design of a medium for composing partial specifications from event sequences and for filtering, clustering and constraining any event streams that match such a specification.

Chapter 4

The Specification of Parallel Behaviours

4.1 Introduction

In Chapter 2 we described the failings of the few debugging and error handling systems that currently exist for object oriented languages and we briefly covered the advantages of operational specification, in these applications, in Chapter 3. In this chapter, we present a technique, relevant to both debugging and exception detecting, to describe and specify the behaviour of software at the level of individual objects. Our goal is to demonstrate the feasibility of an operational formalism as the basis of a technique for in-source specification, bug hypothesis testing and run-time error handling. We present EPS—*Event Pattern Specifications*[1]—a facility for the enunciation of object behaviour. EPS augments a 'host' object oriented language (or a debugging system) and allows the user to express the desired conduct of each defined object in terms of one or more Event Patterns, using a language based on CSP [Hoa85]. Here, we detail the theory and usage of Event Pattern Specifications and later, in Chapter 5, we explain how EPS specifications can be used to improve object reliability as the basis of a unified, behavioural exception handling mechanism. An implementation of EPS is described in Chapter 6.

4.2 The Benefits of Operational Specification

The need to enunciate the correct behaviour of an object, as explained in Chapter 2, implies the use of a specification language. However, attempts to use an established specification language like

[1] In previous publications, e.g. [PW90, PW91b], these were known as MPS—Message Pattern Specifications. The name change reflects minor semantic changes made since then.

OBJ [Shu89], Z [Spi89] or Clear [BG81], or an object oriented equivalent, would be impractical and inappropriate. Such languages are designed to perform verification at the design stage of software development, not to monitor or describe parallel run-time behaviours. Ideally, a means of expressing behaviour using full predicate calculus or temporal logic should be provided. Alas, this too is beyond the realm of current pragmatism [Mey88, KJ88, CW89, Fid89]. Indeed, any method of specification that concentrates on internal form and semantics is likely to be non-viable because of the dearth of popular, mathematically rigorous, parallel object-oriented languages. Instead, we use a language-independent method based on object behaviour.

Operational specifications are expressed solely in terms of the events undertaken by objects. Object behaviour, in a parallel system, is specified by a partial ordering on this sequence of events. The power of this method is chiefly responsible for the wealth of event-based debugging and monitoring techniques that exist for parallel and distributed systems [Bat87b, Bat89, BH83, LL89, Smi85, BLW89]. Event-based models of behaviour may be expressed at many levels of abstraction and event filtering may be used to support program slicing [Wei82], a behavioural analysis technique. Furthermore, such models are entirely independent of the source language and support concurrency with relative ease. Several concurrent calculi exist about which a operational specification language can be built; CSP [Hoa85], CCS [Mil89] and LOTOS [Bri87] are but a few obvious candidates.

4.3 Design Requirements

The design goals considered here are consistent with a desire to build a specification notation that is capable of concise, easily built and readable specifications. We sought to develop a powerful specification medium that could be easily understood by users. This reflects the duel rôle of EPS as a specification and a documentation tool. In addition, the requirements must take into account the environments in which EPS will be used. Two such environments are discussed: using EPS as an exception signalling mechanism in a parallel object oriented language to improve the behavioural rigour of objects; and its use within a debugger to directly support hypothesis confirmation and complex context generation. Each environment has considerations, and therefore functionality, unique to it. However, the vast majority of EPS's design decisions are pertinent to both.

4.3.1 Common Design Requirements

For the purposes of this discussion, we can view each specification as a set of rules for a parser. At run time, each such parser 'sees' the stream of events representing the behaviour of the object(s)

with which they are associated and attempts to satisfy its rules in the light of this observed behaviour. In fact, this viewpoint is a simplification and the its limitations are discussed in Chapter 6. Parser rules need to be expressed in a small, rigorous and succinct grammar, capable of describing *any* partially ordered, juxtaposition of primitive events that constitutes a possible concurrent behaviour. The need for a formal grammar with well understood semantics can be satisfied, as suggested above, by using an established event calculus as the basis for its design. The grammar should be easy to understand and easy to transcribe to a graphical medium to facilitate visualization.

To satisfy these demands and to match the flexibility of the features discussed in our taxonomy of event based systems (see Section 2.4.11), we require a grammar that facilitates:

- **Comprehensive Specification.** Allowing users to specify multi-threaded behaviour with events *and* state assertions;
- **Partiality.** Allowing the user to imbue her specifications with varying degrees of partiality, in order that the behaviours of the object can be partitioned with some generality;
- **Specialization.** Permitting specifications to be limited to one facet of an object's behaviour;
- **Readability.** Enabling specifications to serve their second rôle as 'active documentation';
- **Reactivity.** Enable each specification to trigger activity on its success or failure; and
- **Reusability.** Permit specifications to be used flexibly in more than one context.

In the sections that follow, each of these requirements is justified and elucidated.

4.3.2 Comprehensive Specification Medium

It is clear that to support concise specifications, the grammar should be capable of expressing any aspect of the behaviour of an object oriented system. This requires an event class set (such as that developed in Chapter 3) that fully and unambiguously captures all primitive aspects of object behaviour and a set of constructors that can build specifications from these events. Such constructors should be able to describe the full gamut of temporal relationships between the events: from the sequential behaviour of methods, to the synchronization of separate threads active within an object. Using them, one should be able to assert temporal (both sequential and concurrent) relationships between processes and methods, at a level of abstraction close to the problem domain.

In Chapter 3, we established that the behaviour of an object oriented system could be completely specified by a stream of just eight types of control event. However, program execution is a dichotomy of actions (control and data events) and state. Although one can specify behaviour exclusively in one medium, it is more elegant and powerful to use both. This implies that some provision for state assertions would enhance the elegance of our grammar. The potential disadvantage of such annotations is that they could introduce, in the specification, dependencies on the implementation of the host object that prevents them being used as axioms for an abstract data type and limits their reuse potential. This can be avoided if state checks are granted no special access privileges to the implementation, beyond these enjoyed by other objects. However, there are occasions when such privileges are useful, e.g. while debugging.

Coupled with partial specification and unification (see Section 4.3.3), state assertions allow one to specify how an object's state alters as a result of method execution or components thereof. This usage of state assertions within behavioural specifications subsumes the functionality of preconditions and postconditions.

4.3.3 Partial Specification

The requirement for partial specification arises from the need for specifications to partition sets of behaviour, with some generality, without resorting to a lengthy series of total specifications. Relying only on total specifications would cripple the expressive power of a grammar. Typically, partiality should enable a set of events, or event interleavings, to be specified in one expression. Its provision implies a requirement for unification, to permit analysis of how the partial components of a specification were satisfied by a matching behaviour. This information, called the *particulars* of a match, precisely locates the matching behaviour within the set of behaviours partitioned by the original specification. Particulars also constitute useful parameters for constraints (see below) or action clauses (see Section 4.3.6) which respectively strengthen the specification and *handle* its violation.

To enhance the power of specification, some behavioural calculi allow additional constraints to be expressed on subtraces of behaviour, or other particulars, acquired from unification [Hoa85, Bri87]. This increases the elegance of specification, as it allows the user to define and partition the behavioural patterns of an object using inequalities in addition to pattern matching. Furthermore, constraints provide a hook via which boolean and temporal algebras may be introduced [Bel89]. They can also help to improve efficiency: since they need not be checked until a successful pattern match occurs, one can defer all computationally expensive calculations by placing them

within constraints. One disadvantage of using constraints is that, because they are not completely orthogonal to pattern matching (both techniques define a set of behaviours), there will be some behaviours that can be specified in two or more ways. Indeed, virtually all specifications that use pattern matching can be re-expressed using a simpler pattern, unification and constraints—with some loss of brevity.

4.3.4 Specialization

As partial specification attempts to enhance the range of behaviours covered by a specification, specialization seeks to reduce or refine it. Users are seldom simultaneously interested in all aspects of an object's behaviour. Focusing attention on behaviour pertaining to a certain activity is often necessary to avoid overwhelming users with information—hence the requirement for specialization. It represents a natural 'selective' approach to the assimilation of related information. Note the distinction between *strengthening* a specification and *specializing* it; the former restricts its likelihood of satisfaction whereas the latter limits its applicability. To support specialization, we require a means to filter the scope of a specification such that it applies only to one aspect of an object's behaviour. To promote modularity, the applicability of a specification should be subject to alteration without any need to alter the parser rules themselves. One flexible way of achieving this is to require that the grammar have some separate facilities to manipulate each parser's input event stream.

4.3.5 Readability

There are few formal requirements that guarantee readability and ease of use—except that each implies the other. Keeping a grammar small makes it easier to use as there is less for the user to remember. Some means of associating specifications with semantic information like names or comments is also required—both as a means of *labeling* specifications for manipulation (i.e. searching, identification and grouping) and for increasing readability. However, labels can not be checked for genuine aptness or relevance and can consequently be misleading. Specifications themselves, however, can be checked and this, whilst not effecting the case for both comments and naming, does imply the need for a self documenting grammar.

It has been established that debugging is largely a problem of comprehension (see Section 2.4.1). Successful specification media must aid comprehension, both of themselves and that which they specify. It is to the advantage of both to support common human complexity management techniques like *chunking*. Humans 'chunk' [Mod79] in an attempt to cluster primitive elements of

behaviour into one atomic activity at a higher level of abstraction. A specification medium should offer some means of supporting this by allowing specifications to be reused in defining more complex, *higher order* ones. This hierarchical abstraction greatly aids understanding, reduces the complexity of specifications and avoids repetition of commonly used sub-specifications.

4.3.6 Reactivity

It is required that these specifications are capable of reacting to their violation. Consequently, when a specification fails (or even succeeds) a wide choice of responses should be available to the checking mechanism. Merely to report all violations is not adequate. The failure of a specification should be capable of triggering any effect that can be achieved within the environment in which the specifications appear. However, it is important that this flexibility of response is not gained at the cost of complicating the syntax or semantics of the grammar unduly. To remain useful, the grammar must be kept small. Consequently, the *action clause* of a specification (wherein its responses are defined) should derive its power by delegating to the support environment, rather than providing functionality of its own. In a debugging context, this might include initiating a sequence of debugger commands and, as part of an exception system, executing a handler method.

Inevitably, some useful action clauses cannot be derived from the support environment alone. Particularly those that manipulate entities introduced by the grammar, e.g. the alteration or querying of the state of other parsers. This is required to enable 'cooperation' between specification parsers. It must be supported by adding the required functionality to the host environment, or by imbuing all specifications with an action clause syntactic component.

4.3.7 Reuse

To achieve a high degree of usability, specifications must be easy to find, understand and re-apply. To enhance the first of these, specifications should be assembled into persistent collections that may be classified, browsed and searched through using database operations or conformance analysis. How easily specifications are understood is determined by their readability, which is already a requirement (see Section 4.3.5). Re-application, the most demanding requirement, dictates that the specification medium is sufficiently abstract to enable specifications to be reused in behaviourally analogous situations—without significant alteration. This might require the standardization of message protocols of analogous data structures, but this is far from a disadvantage, and is catered for directly by some variants of inheritance.

4.3.8 Debugging Design Requirements

In its obligation as an aid to debugging, the grammar should have functionality targeted at enabling greater understanding of the target and supporting hypothesis testing and confirmation. The latter is an important part of debugging that is currently rather poorly supported by debugging tools (see Section 2.5). In support of these activities, the action clause of the grammar (see Section 4.3.6) will have to support some additional facilities that cannot be delegated to the debugger. Two notable examples involve the representation of behavioural traces for which most debuggers have no facilities. These are: the description of how a partially specified behaviour was satisfied (i.e. the *particulars* of a match, see Section 4.3.3); and an important variant—the 'difference plot', an explanation of how the actual behaviour deviated from the specified behaviour to cause a violation. Both are used to visualize behaviour. Also, an action clause that results in program suspension is required, although this might be difficult to implement in a parallel environment (see Section 2.4.13). Using this clause, a specification can be used to establish a debugging context at the point of match or mismatch—it is effectively a behavioural breakpoint. Because many of the specifications will be created 'on the fly', the grammar should optimize the semantic content of specifications while minimizing the cost of composition in preparation time, short-term memory and keystrokes. Such a cost should not greatly exceed that of specifying a lexical breakpoint in a conventional debugger.

The formalism should support the enunciation of patterns of behaviour involving multiple objects, some of which may not have activation records associated with them, or even exist at the time of specification. This implies the need for a nomenclature to uniquely identify objects that *will* exist at some time in the future, but may not currently. Also, this mechanism will, in naming a class, need to distinguish between the unique class object and the notional *wildcard* (i.e. that which matches any instance of a given class).

Often, as with lexical breakpoints, two or more specifications may be used together to ensure a compound behaviour difficult to specify using one alone, even with the benefits of hierarchical definition. On other occasions some way of associating a set of specifications is required as, unlike the exception handling context where they belong to objects, they have no organizing structure when used within a debugger. In lieu of parent objects, these grammars should have some concept of groups: allowing sets of specifications to be addressed as one entity in a relationship outside that of hierarchical inclusion (see Section 4.3.5).

4.3.9 Exception Signaller Design Requirements

Specifications are constraints on the behavioural concretization of abstract data types. They detect, or *signal* exceptions when these constraints are violated. They need to be able to represent the seat of all error checking within the object, leaving the method code clear of *sanity checks* and other verbosities. This separation allows each method to be implemented using only the code that derives directly from its formally rigorous description, even if this method has a partial domain or range.

The addition of specifications to objects is an attempt to improve the exactitude of programs through enhanced accountability, to improve the readability of class definitions and to make systems more 'debuggable'. The approach requires a three tier object design strategy, as opposed to the traditional, two tier approach [Pun90] in which signatures and then implementations are considered separately. Object specifications introduce a behavioural phase, betwixt the other two, such that the design considerations for each object type become:

- **Signature**. To what methods can the object respond? What types of arguments and results are involved?
- **Behaviour**. What are the (parallel) behavioural constraints of the methods presented? What axioms relate the method set?
- **Implementation**. How does each method achieve its ends?

In addition, specifications must be capable of constraining the parallelism available to object methods. They emphasize the fact that, for objects in a parallel system, the *shopping list* criterion [Mey88]—which asserts that all methods of an object may be applied at any time—does not always hold. The need to think about the usage patterns of a object to formulate specifications, enhances the user's model of that object and reduces the probability of error. It also allows object accountability to be less reliant on state based checks and instead to use behavioural checks which are at a higher level of abstraction and, because of their independence of object implementation, more open to reuse between objects with analogous semantics.

The design of specifications should be congruent with the object model of the host language, with a need for as few extra concepts as possible. Unlike the debugging context, the state based elements of the grammar must not violate any language principles, for example encapsulation or locality.

It is prudent to discuss some of the exception handling mechanism design issues covered in Chapter 2 here, notably those of signaller placement and association (detection, linkage and handling regimes are further discussed in Chapter 5). To faithfully reflect the concept of axiom, it is required that specifications are defined (placed) as part of a class description and are associated with all instances of that class. Although it is advantageous for specifications to be able to isolate the behaviour of a particular method, they cannot directly be associated with it *per se*. Since all expressions in pure object oriented languages (except raw assignment) are composed of message sends, there is no need for a more finely grained handler association than this.

Another feature raised in Chapter 2, that of passing parameters from the signaller's environment (i.e. from the action clause and violation context) to the exception handler, so that some of the context of the exception is visible from the handler, is a desirable feature. It allows more generic exception handlers that can individually correct an entire class of related problems, the particulars of which are passed as arguments. This requirement must be weighed against that of congruency with the host language. For were parameterized action clauses to be introduced, the host language would need to be capable of manipulating *all* of the entities that are available to the specification environment, e.g. events, traces, sets... (see Section 4.4.3). To avoid this, and to ensure that encapsulation is not violated, parameterization needs to be designed such that only aspects of state visible to the object owning the specification may be passed. Such a design will need to avoid implicit language dependence.

4.4 The Design of Event Pattern Specifications

4.4.1 Primitive Aspects of Behaviour

Before the behaviour of objects can be operationally specified, the event alphabet in which they indulge must be determined [LL89]. Through the creation and analysis of a parallel, object oriented system model, in Chapter 3, we have derived this fundamental alphabet of *primitive events*. The model demonstrates that, in such a system, the entirety of object behaviour can be expressed as a sequence of events of only eight classes. Defining and minimizing this alphabet helps to satisfy the requirements for a small grammar with well defined semantics (see Section 4.3.1). Each event class and the attributes (parameters) that distinguish individual instances are described below, with the *primary attribute* first. All events have, in addition to the explicit attributes listed below, the implicit attributes of time (i.e. when they occurred) and process identifier.

- **Object Allocation.** The event class *create*, parameterized by the newly created instance.

- **State Access.** The event class *access*, parameterized by the instance involved and the type of access (read or write).
- **Object Destruction.** The event class *destroy*, parameterized by the instance destroyed.
- **Message Send.** The event class *send*, parameterized by the selector, its arguments, the sender, the recipient and the category of synchronization used by the communication (see Section 3.3.3). This is the only event class parameterized by references to multiple object instances; consequently it can be used to facilitate inter-object specifications.
- **Method Execution Start.** The event class *execute*, parameterized by method name, host instance and defining class.
- **Method Execution Termination.** The event class *terminate*, parameterized by method name, host instance and defining class.
- **Delegation (or Superclass Lookup).** The event class *lookup*. Instances of this event occurs when a selector from an incoming message fails to match any method in the local dictionary, resulting in it being forwarded to the proxy (or superclass). It is parameterized by selector, arguments, original recipient and proxy.
- **Error.** The event class *error*. This event covers a multitude of system failures and exceptional behaviours. For example, an error event is generated if, during an attempted selector lookup, the root of the inheritance hierarchy has been reached without a successful match. It is parameterized by the name of the error, the method name, the host instance and the host's defining class.

All object behaviour constitutes sequences (or parallel compositions thereof) of instances of these eight event classes.

4.4.2 Overview of EPS Semantics and Syntax

An EPS is a modular specification of the operational behaviour of an object. It defines a 'valid' set of event interleavings for the object with which it is associated and, optionally, procedures to follow if that object should succeed or *fail* this specification. An EPS is said to *fail* if *tr*, the trace of all events actually exhibited by an object, does not conform to the behaviour pattern the EPS describes. Each EPS consists of five parts:

- **Labels.** All specifications are labeled with user-defined names and comments to enhance their readability. Names are essential in identifying those specifications that fail at run-time and as 'handles' though which to identify individual (or groups of) EPSs for various reasons;
- **The Relevant Trace.** The Relevant Trace expression allows the input event stream of the EPS parser to be manipulated *before* the specification parsing begins. Essentially, it defines the subtrace of *tr* pertaining to the specification and thus permits modular specialization (meeting the requirements of Section 4.3.4), in much the same way as a generator does in a set comprehension [Spi89]. Many specifications can be simplified by local filtration of the irrelevant events from the stream *tr*;
- **The Main Specification Template.** The ordered event pattern to which *tr* must conform, in order to satisfy the specification. It is constructed from primitive events and temporal operators. Partial specifications can use wildcard traces to express their partiality. These wildcards may be named, bound variables. If *tr* conforms to the event pattern, these named variables become instantiated with those *particulars* they represent (see Section 4.3.3).
- **Additional Constraints.** These are optional conditions that strengthen a specification. They are analogous to the postconditions in a set comprehension. Expressed in a language similar to the specification clauses of CSP [Hoa85], these clauses often express further specification requirements in terms of the variables bound by unification (see above).
- **Action Clause.** This is the response of the specification—the behaviour that is triggered if the specification succeeds or fails. The exact nature of the response is dependent on the environment in which EPS are used. This is an optional component of each specification and a default behaviour is triggered if an action clause is not provided.

The overall EPS syntax is:

Name Comment.	
[**groups**	Grouplist]
satisfies	Relevant Trace
inwhich	Template
[**iff**	Constraints]
[**then**	Action Clause]
[**else**	Action Clause]

4.4.3 Types

Before examining the components of event pattern specifications in detail, the types of data manipulated by EPS are discussed. EPS operators and constructs act on a variety of special data types.

Some of these are already available in the language (or debugging) environment, specifically:

- **Numerics**. For example, integers I, reals R;
- **Alphanumerics**. For example, strings S;
- **Booleans** (B);
- **Process Identifiers** (P); and
- **Object Identifiers** (O). Precisely how object references are specified is implementation dependent. Typically some composite means is used to uniquely identity objects (e.g. name and address, or name and value), although the user may not perceive all fields of this composite (see Section 6.4.1).

Some types are not available in the host language. EPSs (internally) use five data types uncommon in programming languages or debugging environments:

- **Event** (E). Events are n-tuples consisting of an event class specifier (one of the eight classes named in Section 4.4.1) and the attributes they require. For example: `lookup(abs, (), real, numeric)` indicates that a search of the selector *abs* (with no arguments, hence '*()*'), failed in the dictionary of the class *real* and is passing to that of class *numeric*. The high level of abstraction helps to satisfy the goals of readable, well-formed specifications at a level of abstraction close to the problem domain (outlined in Sections 4.3.2, 4.3.5 and 4.3.8). It is important to note that objects indulge in events, EPS parsers and filters manipulate them and constraints may compare them, but users form specifications from *templates* (see below) *not* events.
- **Trace** ([E]). Traces are sequences of events; a list of events in temporal order, with the semantics of CSP traces (see [Hoa85]). The trace *tr* represents the entire behaviour of the host object and is the default input stream for all EPSs. Syntactically, where they appear in constraints, literal traces are sequences of events separated by commas and delimited with angled parentheses '<>'.
- **Alphabet** ({E}). Alphabets are sets of events which have the semantics of CSP alphabets (see [Hoa85]). They represent the event repertoire of an object or trace. Syntactically, they are identical to traces, except that they are delimited by curly braces '{}'.

- **Template Particle** (TP). These are the atoms of specification templates—each matches with one event in the input stream. They have a syntax that is a superset of the event tuple, shown above. In addition to the event syntax, template particles may introduce partiality through the use of abbreviation, or wildcard operators. Using abbreviation, one may specify an event using only the type and primary attribute, or just the former. Hence the example event given above could be specified by `lookup(abs)`, or just `lookup`. Three types of wildcard exist: anonymous, value and trace wildcards. The anonymous wildcard '?' can replace any argument of a particle, relaxing the match requirement with respect to that component of the tuple. The anonymous wildcard can also replace an entire tuple, yielding a particle that matches any event. Value wildcards consist of a '$' followed by a name. They may be used in place of tuple arguments and act as a wildcard; when the particle is matched, they adopt the value of the argument they represented. Trace wildcards (which have the same syntax) also unify on a match, but they represent entire event tuples or traces thereof.

- **Template** (T). A specification template is a series of one or more particles bound by temporal and state operators. Templates form the core of an EPS; they define a legal partial ordering of events. The syntax of a template is a variant of that for traces. Temporal partialities are introduced by various operators and template wildcards that help to relax a specification. The anonymous wildcard '*' matches any sequence of events and trace and value wildcards are available as above.

Instances of each of these types are manipulated solely within the confines of EPS expressions and have no effect on the host language or debugging environment itself. These types are small in number and many are derived from CSP—satisfying the requirements of Section 4.3.1. How instances of these types are used is defined in the sections that follow.

4.4.4 Name

Names are the labels, provided by the user, by which EPSs are identified by and to the system (thus meeting the early requirements explained in Section 4.3.5). In addition, the user can provide a comment which extensively details the purpose of the specification. To support association of cooperating specifications (in the debugger environment only, as justified in Section 4.3.8), each EPS has a group list which enumerates those groups to which it belongs. Several predefined groups exist: *active*, *matched* and *unmatched* respectively contain those EPSs that are active, those that have ever matched and those that are being parsed but have so far failed to match (see Section 4.4.6). It is important to realize that membership of the latter two groups is *not* mutually exclusive. The user may create her own groups merely by adding a name to this list.

Operator		*Semantics*	
Syntax	*Type*	*CSP*	*English*
tr `restrict` ℓ	$[\mathsf{E}] \rightarrow \{\mathsf{TP}\} \rightarrow [\mathsf{E}]$	$tr \restriction \ell$	yield subtrace of tr containing only the events (or classes) in the set ℓ
tr `eliminate` ℓ	$[\mathsf{E}] \rightarrow \{\mathsf{TP}\} \rightarrow [\mathsf{E}]$	$tr \restriction \{\alpha(tr) \setminus \ell\}$	yield subtrace of tr containing all but the events (or classes) in the set ℓ
`head` tr	$[\mathsf{E}] \rightarrow \mathsf{E}$	tr_0	the first element of tr
`tail` tr	$[\mathsf{E}] \rightarrow [\mathsf{E}]$	tr'	the first element of tr is removed

Table 4.1: Functions Used to Generate Relevant Traces

4.4.5 The Relevant Trace

The relevant trace is an expression in terms of tr, the total behaviour of an object, that denotes these aspects of behaviour that are of interest to a particular specification. Filtration is an essential feature of event monitoring systems [Bat89] (see also Section 4.3.4), which enhances their ability to specify system behaviour at different levels of detail and specialty. Filtration avoids the consideration of irrelevant events, improving the efficiency of man and machine. Relevant trace expressions yield sub-traces that can be matched against the specification template.

The simplest relevant trace is tr itself, indicating that the entire behaviour of the object will be considered by the specification. Filtering may be achieved though use of the tr `restrict` ℓ construct, which restricts the relevant trace to events (or event types) contained within the set ℓ. This function has semantics identical to the CSP operator '$\restriction$'. Similarly, tr `eliminate` ℓ, filters all the events in the set ℓ out of the trace tr. Other operators include those listed in Table 4.1.

For example, the relevant trace for an EPS belonging to a stack object might be:

```
(tr restrict {execute(push), execute(pop)})
```

This is the subtrace of tr representing all *push* and *pop* operations on the stack. Having limited the view of the specification to these operations, we can go on to specify (in the main template) that the number of *pop* operations should never exceed the number of *push* operations. Similarly:

```
head (tr restrict {lookup(foo,(),?,?), error(doesNotUnderstand,foo,?,?)})
```

is a stream of just one event: the first delegation of the incoming selector *foo*. The specification may then, for example, establish constraints on the proxy for this method.

4.4.6 The Specification Template

The main body of an EPS is a template, constructed from instances of the primitive event classes (template particles) using pattern operators. It serves an analogous purpose to that of a syntactic regular expression in LEX [LS75], with the exception that, out of necessity, the syntax is somewhat different and extended to allow partial ordering and state constraints on events (to satisfy the requirements outlined in Section 4.3.2). Also, unlike LEX, the specification is designed to aid semantic review (see Section 4.3.5): to help users to quickly determine how an object is used and its limitations. These functions are vitally important to software reuse [SBK81].

To ensure some degree of congruence with the event model of Chapter 3, the template language is derived from CSP and adopts most of the power thereof (thus satisfying the expressive power of parallelism as defined in Sections 4.3.1 and 4.3.9). Although it lacks any inheritance operators, the fundamental alphabet of events with which behaviours are specified includes inheritance lookup (see Section 4.4.1).

Each pattern operator takes one or more templates or template particles (ρ) and yields a more complex template from them. The operators are listed in Table 4.2, wherein the process denoted X is a *don't care* term, $\mathbb{N}_1$ is the set of all *strictly positive* integers and the state expression β is a boolean assertion (in the host language) binding the observable state of the object concerned. State assertions are used to support state oriented templates. Encapsulation must not be violated in verifying state assertions (see Section 4.3.2), so the boolean expressions are formed from non privileged message passing statements that have no side effects. Readers may notice that the symbol '*' appears in Table 4.2 representing the repetition operator. This is no way conflicts with its earlier definition (in Section 4.4.3) as the anonymous trace wildcard, because the two representations can be distinguished by context *and* can not be used together.

The simplest template is merely a single particle. For example `terminate(push)`, which signifies the termination of the method *push*. The EPS language supports the specification of sequential and concurrent sequences (`,`, ..., `||`), deterministic and non-deterministic choice (`\/`) and iteration (`*`, `*n`), to compose useful behaviour patterns from these particles (see the requirements in Section 4.3.1).

For example, consider an object representing a screen that has subordinate objects representing things displayed on that screen. A specification to ensure that, in a method *redraw*, the screen is cleared before the flood of redraw messages is sent to subordinate displayed objects might read:

Operator		*Semantics*	
Syntax	*Type*	*CSP*	*English*
$\rho*$	$\mathsf{T} \rightarrow \mathsf{T}$	$\mu X.(\rho \rightarrow X)$	the template ρ occurs an unspecified number of times in succession
$\rho*n$	$\mathsf{T} \rightarrow \mathsf{I} \rightarrow \mathsf{T}$	ρ^n where $(n \in \mathbb{N}_1)$	n successive instances of the template ρ occur
$\tilde{}\rho$	$\mathsf{T} \rightarrow \mathsf{T}$	$x : (\alpha tr - \rho) \rightarrow \ldots$	any template *except* ρ
ρ_1,ρ_2	$\mathsf{T} \rightarrow \mathsf{T} \rightarrow \mathsf{T}$	$\rho_1 \rightarrow \rho_2$	template ρ_2 occurs immediately after ρ_1
$\rho_1 \ldots \rho_2$	$\mathsf{T} \rightarrow \mathsf{T} \rightarrow \mathsf{T}$	$\rho_1 \rightarrow X \setminus \{\rho_2\}; \rho_2$	template ρ_2 occurs after ρ_1
$\rho_1 \backslash / \rho_2$	$\mathsf{T} \rightarrow \mathsf{T} \rightarrow \mathsf{T}$	$(\rho_1 \rightarrow X) [] (\rho_2 \rightarrow X)$	template ρ_1 or ρ_2 occurs
$\rho_1 \| \rho_2$	$\mathsf{T} \rightarrow \mathsf{T} \rightarrow \mathsf{T}$	$\rho_1 \| \| \| \rho_2$	the templates ρ_1 and ρ_2 are interleaved
$\rho\#n$	$\mathsf{T} \rightarrow \mathsf{I} \rightarrow \mathsf{T}$	$\|\|_{0<i<n} \rho$ where $(n \in \mathbb{N}_1)$	n interleaved occurrences of the template ρ
$\rho/[\beta]$	$\mathsf{T} \rightarrow \mathsf{B} \rightarrow \mathsf{T}$	$\rho \not< \beta$	as the template ρ occurs, the state β is attained

Table 4.2: Pattern Operators Used to Generate Specifications Templates

```
EPS: FinishClear
''Ensure screen wipe is complete before re-drawing child windows.''
```

```
satisfies  tr restrict {send, execute, terminate}
inwhich    execute(redraw),execute(wipe),
           terminate(wipe)...(send(redraw))*,terminate(redraw)
```

In many specifications, there is a need to distinguish instances of the same event occurring within partially concurrent, *overlapping* threads active within the same object (see Section 4.3.2). This is achieved by labeling events with value wildcards which are unified with the process identifier (an implicit attribute of all events, see Section 4.4.1) at run-time. Such labeling is accomplished through use of the optional operator ':'. Hence to indicate a sequence of events *send*, *lookup* and *execute*, in which *send* and *execute* are generated by the same thread i, one may use the template:

```
$i:send, $j:lookup, $i:execute
```

Here, after the occurrence of an event such as `1201:send(redraw,(),foo,bar,sync)`[2], the value wildcard `$i` is instantiated with the value 1201. Consequently, for the template to match

[2] We assume in this example that process identifiers are represented as a series of contiguous, uniquely attributed integers.

the final event, *execute*, it must also have a process identifier of 1201. Note that `$j` may or may not equal `$i`, since it is not used again it could be replaced with the anonymous wildcard '?' or the entire ':' label omitted altogether. Both are instances of *unified value wildcards*. To illustrate their value, consider a stack object. In order to ensure serialization of the *push* method (i.e. to ensure that the method *push* can hold at most one thread), one might use a specification:

```
EPS: PushMonitor
''Ensure that only one thread enters push at a time.''

satisfies  tr restrict {execute(push), terminate(push)}
inwhich    $i:execute(push), $i:terminate(push)
```

When an object is created and its EPS 'parsers' are initialized, all templates are *pre-dormant*, i.e. they are neither matched or unmatched. As the host instance indulges in events, its event stream is fed to each parser (through their respective relevant trace filters) as shown in Figure 4.1. Unless the EPS is *pre-strict* (i.e. it begins with a period '.'), the parser will ignore all events in the relevant trace until one occurs that confirms to the first particle in the template. When this occurs (or with pre-strict templates, after the receipt of the first relevant event), the parser becomes *active*. As further relevant events become available, the parser advances through the template. Eventually, the EPS will fail because one relevant event will not conform to the template, or it will succeed as the end of the template is reached. If the EPS fails, it is marked as *unmatched* (it joins the group of the same name) and the `else` action clause is acted on. If it succeeds, it is marked as *matched* and the `then` clause is taken. If the matched template is *post-strict* (i.e. it ends with a period '.') the EPS becomes *post-dormant* and is never checked again, however non post-strict templates are reset to their starting condition after a match.

Templates starting with an unspecified particle, i.e. those that begin with the anonymous wildcards `?`, `*`, or with a named trace variables, must be pre-strict. Similarly, those ending with a wildcard that is not *right bound*[3] must be post-strict (since they can never terminate). Indeed in the latter case the template, if satisfied to the point of the final wildcard, will enter a *perpetual match* condition. Each new event that arrives will be appended to the growing tail sequence matching the wildcard, but the end of the template will never be reached. This, apparently baneful, condition is sometimes useful—especially when combined with constraints as shown below.

[3] A wildcard is said to be right bound if it has a defined end point, i.e. there exists a condition whereby the wildcard will be satisfied and its parser will attempt to satisfy the particle that follows it.

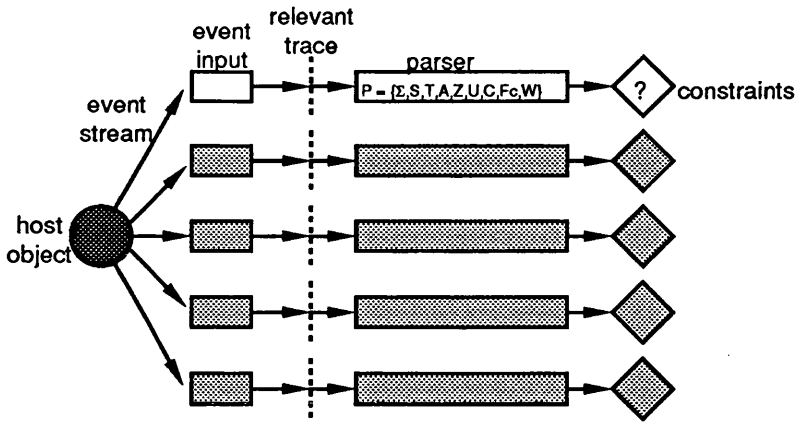

Figure 4.1: Distribution of Event Stream to EPS Parsers

4.4.7 Additional Constraints

Relevant traces and specification templates are not enough to describe all behaviours (see Section 4.3.3). They can filter and specify the ordering of permitted events, but are unable to express the inequalities that often comprise real-world specifications. Also, the partiality introduced by wildcards and the operators '...' and '||' cannot be constrained as part of the specification—a further limitation. To overcome these limitations, an optional constraint clause can be used to supplement EPSs. Much of the expressive power of CSP can be added to EPS through constraints.

Using constraints, any partially specified sections of the template, on which it is desirable to impose further constraint and that hitherto had been represented with wildcards or operators '...' or '||', is instead denoted by a named trace wildcard. Similarly, any tuple argument previously represented as '?', may be replaced by a named value wildcard. Then, after matching, if *tr* conforms to the template, these variables are instantiated with the traces or values they represented. The constraint clause may then strengthen the EPS by expressing additional conditions for its acceptance in terms of these traces, values and their derivatives. Table 4.3 depicts the operators that constraints use, in addition to these shown in Table 4.1, to manipulate trace variables or calculate their properties.

The traces and alphabets generated by constraint functions (through the use of trace variables and the operator '@') can be tested using a range of set operators including `in`, `subsetof` and `equals` (which have the semantics of the mathematical relations $\in$, $\subset$ and $=$ respectively) and their negations. They may be compared with other traces or alphabets, or with literals like <>, the empty trace, and {}, the null alphabet. Trace and alphabet lengths (found using #) and

Operator		*Semantics*	
Syntax	*Type*	*CSP*	*English*
$tr[n]$	[E] → I → E	$tr[n]$ where $(n \in \mathbb{N}_1)$	the n^{th} element of trace tr
$tr[n...m]$	[E] → I → I → [E]	$\wedge_{n \leq i \leq m} tr[i]$ where $(n, m \in \mathbb{N}_1)$	a subtrace a tr consisting of the n^{th} through to the m^{th} element of tr
$@tr$	[E] → {E}	αtr	the alphabet of trace tr
$\#tr$	[E] → I	$\#tr$	the length of trace tr
$tr!\ell$	[E] → E → I	$\#(tr \upharpoonright \ell)$	the number of occurrences of ℓ within tr
$tr_x/\backslash tr_y$	[E] → [E] → [E]	$tr_x \hat{} tr_y$	trace (or event) concatenation

Table 4.3: EPS Constraint Functions

instantiated, named, value wildcards can be likewise compared using standard magnitude relations (<, >= ...). In a constraint expression, the semantics of any relation may be negated by prefixing it with the symbol '~'.

Additional constraints are so powerful that some specifications may be expressed using them alone. For example, to ensure that an initially empty stack is never requested to *pop* when empty, one might use the EPS:

```
EPS: NotEmpty
''Ensure that pop is not called more often than push.'~

satisfies  tr
inwhich    $a
iff        $a!{execute(push)}>=$a!{execute(pop)}
```

This is a simple example of a template that is not *right bound.* That is, since it ends in a wildcard it can not terminate until it fails or until the associated object is destroyed. This example is extreme in that it is also *pre-strict* and will consequently enter the *perpetual matching* condition immediately. In this condition, the template matches (and the constraint is checked) as each new event occurs. Although this is not the most efficient method, it alleviates the need for a method precondition in this case. Assuming the existence of a method *size*, another specification achieving the same goal is:

```
EPS: NotEmpty
''Ensure that pops are only conducted on non-empty stacks.''

satisfies  tr
inwhich    ~execute(pop)/[(self<-size) <= 0]
```

Where a<-b denotes a message send to object *a* of selector *b* (as in Smalltalk-80 [GR83]). This message send is synchronous, for obvious reasons, and only *accessor* methods[4] should be used in this way. Postconditions may be simulated in EPS in a similar fashion to the precondition above. For example, to ensure the *push* method increases stack size:

```
EPS: PushGrows
''Ensure that after a push the stack has grown.''

satisfies  tr restrict {execute, terminate}
inwhich    $i:execute(push)/[$1 = (self<-size)]...
           $i:terminate(push)/[$2 = (self<-size)]
iff        $2 > $1
```

Here, the state operator '=' forces the value wildcards $1 and $2 respectively to be instantiated to the values yielded by the message sends. For an example of the usage of trace variables, consider that for a window object to assert that it should be moved only when opened and destroyed only when closed, one may specify:

```
EPS: MoveWin
''Ensure window can be manipulated only whilst open.''

satisfies  tr restrict {execute, destroy}
inwhich    (execute(open)...execute(close), $z)*,
           destroy
iff        execute(move) notin @$z
```

In many cases one can express the same specification semantics using a complex template or a simple template with a constraint. Often, although the choice between the alternatives may seem hard, there are subtle reasons for choosing one method as opposed to the other. Consider an object capable of sending messages *a*, *b* and *c*. Consider the two specifications given below to ensure that *b* is sent at least four times between the sending of *a* and *c*.

[4] Accessor methods relay some facet of the receiving object's state, without changing that state [Mey88].

```
EPS: Protocol1
''Ensure b is executed the minimum number of times.''
-------------------------------------------------------
satisfies   tr restrict {send}
inwhich     send(a),send(b)*4, send(b)*, send(c)
```

```
EPS: Protocol2
''Ensure b is executed the minimum number of times.''
-------------------------------------------------------
satisfies   tr restrict {send}
inwhich     send(a),send(b)*$n, send(c)
iff         n>4
```

The former is more efficient and might be used if efficiency were a prime concern. The latter is more complex, but allows a better and more flexible form of expression which is easier to change. For example, to additionally constrain the repetition according to an argument of the method *a*, one might change the specification to:

```
EPS: Protocol3
''Ensure b is executed the minimum number of times.''
-------------------------------------------------------
satisfies   tr restrict {send}
inwhich     send(a,($x),?,?,?),send(b)*$n, send(c)
iff         n>4 \/ x>=n
```

4.4.8 Action Clauses

No definitive syntax for action clauses is given here, since handling the problem signified by a failing specification is highly dependent on the environment supporting the specifications.

In a debugging environment the useful actions that could be triggered by specifications can be placed into three categories:

- Executing user defined sequences of debugger commands
- Altering the input event stream of associated parsers
- Altering the state of associated parsers

Within a debugging environment such as that provided by GDB [Sta88], DPD [HK89] or Spider [Smi85], all of which support the creation of named sequences of debugger commands (macros), an action clause could invoke such a macro by name. More advanced host debugging systems might support parameterized macros. This type of action clause delegates the functionality to the host environment, allowing EPS to remain environment independent. Through macro calls, EPS can tap the entire functional repertoire of a debugging tool (see the requirements in Section 4.3.6) and use it to facilitate experiments—from printing the value of a variable (thus using the specification as a behavioural filter), to incrementing a debugger maintained counter (to determine how often the specified behaviour occurs). The single, most effective debugger action clause for EPS is the ubiquitous 'stop' directive (meaning halt execution and render the current context open to examination), enabling EPS to be used as a very sophisticated breakpoint facility (see Section 4.3.8).

Another feasible action clause, within a debugging environment, is to append to an object's event stream a user defined, parameterized *higher order* event. In effect, the EPS ceases to be a stand-alone specification and becomes a definition of a higher order event that contributes to the event streams of other EPSs. Other EPSs may then specify this event in their templates, increasing the level of abstraction and readability therein (see Section 4.3.5). For instance, consider that the *read* and *write* cycles of a file object might be defined:

```
EPS: Readcycle
''open a file and read from it.''
-------------------------------------------------------------------
satisfies  tr restrict {send(open),execute}
inwhich    send(open,("r"),?,?,$sync),execute(read)*,execute(close)
then       append read($sync)
else       append access_error("read")
```

```
EPS: Writecycle
''open a file and write to it.''
-------------------------------------------------------------------
satisfies  tr restrict {send(open),execute}
inwhich    ($i:send(open,("w"),?,?,$sync) \/$i:send(open,("a"),?,?,$sync)),
           $i:execute(write)*,$i:execute(close)
then       append $i:write($sync)
else       append $i:access_error("write")
```

Here the action clause **append**, appends the higher order event to the input stream for that object. Notice how the synchronization mode is propagated as a parameter of the new high order events *read* and *write*, and similarly the access type for the high order event *access_error*. To ensure

that a multiple read and exclusive write protocol (MREW) is adhered to by a file, one might use the readable specification:

```
EPS: MREWProtocol
''ensure the file follows the MREW protocol.''

satisfies  tr restrict {read,write,access_error}
inwhich    .read(sync)\/write(?)
```

Note that when a higher order event is used in an append clause it automatically creates a new high order event type. The relevant traces for higher order events usually reflect this higher level of abstraction.

To facilitate behavioural experimentation it can be useful, within a debugging context, to forcibly alter the state of one EPS parser from the action clause of another. Since each EPS is uniquely identified by the object with which it is associated and its name, such alteration is achieved by specifying the nature of the change and the EPS(s) affected. Alterations may include (but are not limited to): disabling, enabling, enforced success, enforced failure and termination of individual parsers or groups thereof. For example, whilst debugging a suite of objects that represent a bank account, one may have a suite of EPS-breakpoints belonging to the group *bankdebug*. One of these might be[5]:

```
EPS: EarlyLargeWithdrawal
''breakpoint large transactions occurring early in an account lifetime.''

groups     { bankdebug}
satisfies  tr restrict {send(withdraw)}
inwhich    .send(withdraw,($n),?,?,sync)/[$n<lt(500)]*$m,send(withdraw,(?),?,?,sync).
iff        $m<10
else       large_withdrawal_alert
```

an EPS that fires when a new customer makes a large withdrawal (more than 500 pounds in this example) in the first ten withdrawals. The action clause *large_withdrawal_alert* is a debugger macro that breakpoints the object, allowing the user to examine the details and ensure that the bank object's security features behave as intended. However, these security features may only exhibit a failure after a considerable number of accounts have been processed. It may not be desirable to enable *EarlyLargeWithdrawal* until a large number of records have been processed. One may disable the EPS (from the debugger) and enable it automatically using a second specification:

[5] In the example we assume the existence of the method *lt(x)*, for integer objects, that returns *true* only if the receiver is less that the argument *x*.

```
EPS: EnableEarlyLargeWithdrawal
''enable ELW after 500 operations.''

satisfies  tr restrict {send}
inwhich    .?*500.
then       enable bankdebug
```

The orthogonality of this sort of experimentation is improved if parser manipulation is added to the debugger command language itself.

Command 'macro' sequences and parser alteration are unavailable and inappropriate in the more formal language exception handling environment. Since the specifications model ADT axioms in this environment, the manipulation of parser states is both irrelevant and dangerous, given the invariant nature of axioms. The usefulness of higher order events is somewhat obviated by the existence of stronger structuring mechanisms i.e. object and class hierarchies. The principal exception handling oriented action clause is the invocation of a *handler* method, the EPS in this case is a sophisticated exception definition. However, to name the handler method as part of the specification restricts flexibility and could be considered as a violation of encapsulation (see definition of *linkage* in Chapter 2). In Chapter 5 we define the design and function of an exception handling system built around EPSs.

4.4.9 Persistence and Predefined Events

To aid reuse of EPSs (see the requirements outlined in Section 4.3.7), especially within a debugging context, they may be saved to disc in libraries. Simple search and browsing tools may be used to view saved EPSs to look for reuse candidates. Naturally, these procedures are unnecessary in the language environment where the abstractions are reused, as opposed to the axioms that belong to them.

A number of predefined EPSs exist to aid the formation of frequently used templates. These include those covered in table 4.4. These events differ from higher order events; essentially they are just rewrite rules, but they improve the readability and succinctness of specifications in a similar manner (see Section 4.3.5).

4.5 Visualization

The ease of use of EPS and its effectiveness, especially within a debugging tool, can be considerably improved by careful attention to the way in which behaviour, specifications and the differences

Operator Syntax	*Semantics*	
	Event	*English*
`self(x,arg,s)`	`send(x,arg,y,y,s)`	send a message request to self
`subsend(x,arg,se,r,p,s)`	`send(x,arg,se,r,s)`, `lookup(x,arg,r,p)`	defer incoming message to superclasses
`$i:do(m,i)`	`$i:execute(m,i,?)...` `$i:terminate(m,i,?)`	successfully execute a method

Table 4.4: Predefined EPS Fragments

between them are visualized. The former two are identical in all respects, but the differing visualization requirements of traces and templates. Visualization of behaviour differences must highlight the extent and nature of differences between actual and specified behaviour, which requires a vastly different approach. Thus far, we have only considered the textual representations of behaviour and specifications. Here we discuss briefly the promise and problems of graphical visualization and 'difference plots'.

4.5.1 Graphical Visualization of Behaviour and Behavioural Specifications

Graphical visualization of behaviour is currently an area of intense research [CB86, KG88, Sto88]. There is cognitive evidence to suggest that it aids debugging [DC86, Car83b, Sen83] and although the empirical evidence is somewhat rare [FM89, CBM90], many implementors of debugging software have attempted to use it [BH90b, Bov86, BTM89, CC89, RRZ89, SBN89, Bat87a] in order to improve the ease and enjoyment of using their products. Despite the advantages of graphical visualization, experts often prefer textual modalities for interacting with computers [BEH88], and for some advanced functionalities it is the only medium [BH90a].

Some possible iconic visualizations for template particles are included in Figure 4.2. Clearly, some mechanism to allow user defined icons is required to support higher order events. In addition, some means of iconic combination is needed to visualize templates—straightforward expansion into the icons of Figure 4.2 will produce overwhelming amounts of visual information for non trivial templates. One possibility is the use of *iconic ligatures*, i.e. automatic combination and placement of certain icon sets into single icons, according to a fixed set of rules based on events and their temporal relationships. Figure 4.3 depicts an example of this. It represents the specification templates of the template *PushMonitor* and the predefined EPS *subsend*. In this way a combination of ligatures, user defined icons, placement and special symbols can be used to represent templates. Visualizing the relevant trace, state assertions or EPS constraints is more difficult. Figure 4.4

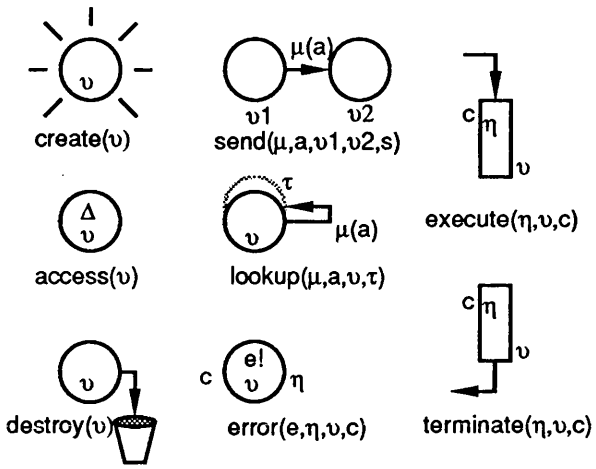

Figure 4.2: Graphical Visualization of Event Alphabet

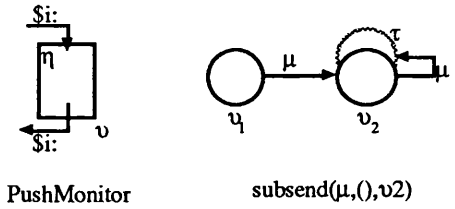

Figure 4.3: Graphical Representation of *PushMonitor* and *subsend* Using Iconic Ligatures

shows one possible technique which relies on textual media to relate the constraints and uses a graphical representation of the specification and synchronization constraints.

Visualizing parallelism is fraught with problems (see the subsection entitled "*Visualization*" in Section 2.4.13), not least because it requires large amounts of screen estate. Abstraction is one solution: event icons such as these listed above can be used to provide detailed representations of behaviour, while traditional concurrency diagrams such as those proposed by Stone [Sto89], Agha [Agh90], Fidge [Fid89], Hewitt [Hew77] or Feldman and Moran [FM89] can be used to describe synchronization. In the computational model to which EPS is tailored (see Chapter 3), parallel synchronization is determined by three types of communication: synchronous, asynchronous and future. In Figure 4.5 we show possible ways of visually depicting these three types of synchronization and Figure 4.4 shows how this might be used within a whole EPS.

Visualization of the potential parallelism permitted by a specification is more difficult than visualizing the actual parallelism exhibited by a live object and is a problem worthy of further work. It is not addressed by this thesis.

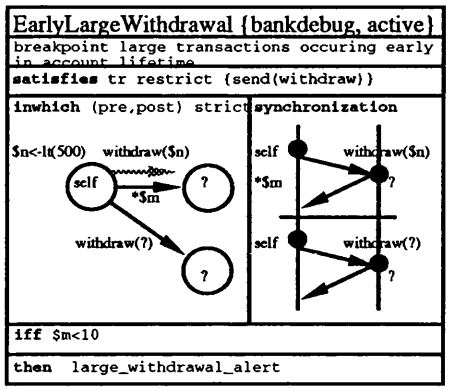

Figure 4.4: Graphical Visualization of the EPS *EarlyLargeWithdrawal*

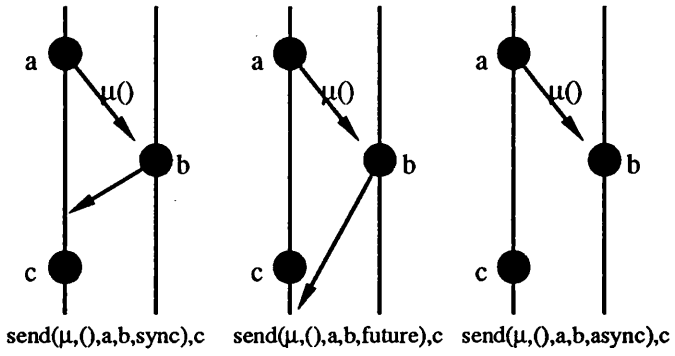

Figure 4.5: Graphical Visualization of Parallelism

4.5.2 Visual Deltas

The culmination of visual specification, especially in a debugging context, is to be able to depict in a high level graphic to what extent the actual behaviour of an object satisfies a specification. That is, to visually demonstrate why—and when—an object's conduct violated its specification. As EPS templates are partially ordered, this cannot be achieved by linear LALR(1) parsing algorithms such as those used to verify YACC input [Joh78] or the *send-expect* sequences of UNIX's *uucico* utility [OT89]. Furthermore, since higher order and primitive events will be mixed, some form of abstraction is needed or the *visual deltas* will convey little useful information. One solution is to use a ongoing conformance test on corresponding nodes of the specification parse tree and thread traces exhibited by the object. This would be especially effective if it were graphically animated. The design of such an algorithm is beyond the scope of this thesis.

4.6 EPS: Debugging versus In-Language Use

Despite the pertinence of the EPS mechanism to both language based axiom support and sophisticated assertions for debugging, some differences exist in the requirements between the two contexts that reflect their disparity. Firstly, there are differences in the techniques used to associate objects and specifications. Within a language context, the association is static and the EPS axioms are provided within a type definition. Within a debugger using EPS, the associations are typically with instances (since it is rarely useful to apply debugging assertions to all instances of a type at one time) which may be uniquely identified by name, context and (when one is specifying the behaviour of an ephemeral instance that may not exist yet) by time or circumstances of creation ([BTM89] for example).

In-language specifications usually only concern the behaviour of the object with which they are defined (although multi-object behaviours can be specified) in order to maximize their potential for reuse. However, assertions formulated within a debugger have no such constraints, behaviours involving more than one named instance may be asserted using the *send* event.

The action clauses available to EPSs in the different environments will differ to a greater extent than any other feature of the formalism. The motive behind and facilities available to the clauses are easily distinguished. In a language context, the goal of the clause is to recover or ameliorate the problem causing a violation of a specification. Within a debugging frame of reference, specifications are used to trap a desired context (using EPS as extended breakpoints) to gather program

information or to alter program behaviour. To avoid incorporating context dependent functionality, EPS action clauses defer much of their functionality to the host environment (as explained in Sections 4.3.6 and 4.4.8) which better facilitates the disparate goals of clauses in the two contexts and accentuates the difference between them.

4.7 Discussion

4.7.1 Advantages

In a debugging context, EPSs are a stronger formalism than alternatives, such as lexical breakpoints or source amendment, for the following reasons: they relate more closely to the problem domain; they have a higher semantic bandwidth; and they are not ad-hoc—forcing some thought about the behaviour of user defined objects on the part of the user. EPS directly supports hypothesis testing and confirmation. EPS allows control and execution contexts to be established that rival methods could not. For example only EPSs can:

- Establish contexts associated with inheritance lookup;
- Ensure that an ADT's temporal protocol is adhered to;
- Actively support hierarchical abstraction;
- Permit direct specification of behaviour without the need to translate into a line of source code;
- Be reused in analogous circumstances with little change; and
- Serve as a documentation aid. This is mainly because, unlike breakpoints, EPS can be easily back-translated into the hypothesis that they were created to establish.

In a language context, EPS offers several advantages over existing exception handling systems, as Chapter 5 will show. Furthermore, EPS is a more portable specification technique than that offered by any exception handling mechanism because it is, although customized for object-oriented languages, language independent. The formalism aids type documentation without violating encapsulation and is amenable to visualization. Use of EPS obviates the need to introduce special variables to reflect key aspects of behaviour, for example the redundant variables *size* and *isEmpty* that are often maintained in stack ADTs and required by state based exception mechanisms.

4.7.2 Limitations

The flexibility and facility of EPSs introduces several problems that can be avoided by careful usage. Ideally, one ought to provide a rewrite mechanism that translates EPSs exhibiting these problems into equivalents that do not. However, such a system would require as much real world knowledge as one designed to formulate EPSs from scratch and is beyond the scope of this thesis.

Because they are asynchronously parsed (and checked) some EPS are immensely inefficient. For example, the first EPS *NotEmpty*(see Page 110) triggers a parsing and constraint check for *every* event reported by stack instances (it is not *right bound*). There is no doubt that such a specification (indeed most non right bound specifications with constraints) would levy a crippling overhead on any system on which it was instantiated. Rewrites such as the second-EPS *NotEmpty* (see Page 111) are generally more efficient, albeit less expressive and reliable (as it depends on the correct function of the method *size* which may itself be flawed).

Some specifications are phrased such that violations can only be realized long after the cause of the problem. This can make recovery difficult and retrieving the correct debugging context impossible. For example, the EPS *MoveWin* will only 'realize' that a window has been sent a move message whilst it is closed when the `$z` trace variable is instantiated by the occurrence of its boundary event (in this case *execute(open)* or *destroy*). Since this is after the occurrence of the erroneous event, the debugging context that can be raised using this specification may not be useful. This can be avoided by putting specifications into the *perpetual matching* condition, that is, constraints are checked during the growth of trace variables instead of just when their instantiation is complete. Unfortunately, this has grave repercussions on efficiency; indeed there is frequently a tradeoff between efficiency and immediacy.

In a debugging context, type checking of EPS specifications could be difficult. In systems with true dynamic binding, type checking specifications will be compromised since, at the time of specification, the type of an object may not be uniquely defined. This poses serious limitations on any static checking of specifications.

Currently, each defined EPS represents one instance of a parser 'listening' to the events of the object(s) with which it is associated. There is no mechanism for instantiating two or more identical parsers, for the same object, from one definition. Furthermore, there is no means of facilitating multiple concurrent parsers that 'look' for the same behaviours in different threads of one object.

Perhaps the most severe problem is that not all EPS definitions make sense or can be parsed. Consider for example the EPS:

```
EPS: Silly
''This EPS cannot parse or instantiate its trace variables.''

satisfies  tr
inwhich    send(x), $1||$2, send(y)
```

There is no *a priori* technique of partitioning the events occurring between the two *send* events betwixt the trace variables. This type of mistake must be trapped by the implementation and flagged as a user error.

Chapter 5

Exception Handling in Parallel Object Oriented Languages

5.1 Introduction

In the previous chapter we introduced the EPS formalism and detailed its general usage as a specification medium and a specific application as a debugging aid. In this chapter, we present a design which grafts an exception handling mechanism based on EPS onto an existing parallel object oriented language, Solve. In doing this, we hope to demonstrate its immediate facility as an exception detection formalism and to investigate the potential of behavioural abstractions for aiding semantic review and searching.

Modern object oriented languages stress the operations that type instances may undergo and the correct argument types and states involved, but offer no concept of correct behaviour patterns. This oversight weakens the potential power of the abstraction (especially in parallel systems) and the ability of such languages to detect behavioural problems manifest at run-time. Without this facility, error detection schemes in these languages—where they exist—are weak and undisciplined.

5.2 Design Requirements of Exception Handling Systems

5.2.1 Purpose

The general purpose of exception handling mechanisms is defined in Section 2.3.1; in summary, they detect and mitigate the effects of method failure, or misuse, at run-time. They support *reliable*

types, instances of which behave correctly within their domain and do not fail catastrophically, or silently[1], otherwise.

Within this context, the purpose of EPS is to define assertions which partition an object's behavioural and state domain, separating conventional behaviour from the exceptional. If the target system is event instrumented, these assertions can be used to detect all forms of anomalous behaviour. In addition, they can supply 'active documentation' which may be used to enhance understanding of types, to facilitate semantic search of a type repository and assess a type's reuse potential in a given circumstance.

5.2.2 Design Considerations

The design requirements of an exception handling system are best analyzed by examining the stages of exception handling introduced in Section 2.3.4: specification placement, exception detection, linkage and handling. One fundamental requirement is that of congruency. In adding an exception handling mechanism, one should perturb the base language as little as possible and add to it as few new concepts as is feasible. Another important consideration is correctness: an exception handling system should never allow software to violate its specifications with impunity.

Placement covers those elements of language design involving the deployment of assertions to check for anomalous behaviour. Ideally, placement should make these assertions unobtrusive, modular, abstract (to promote reuse) and physically separate from conventional code. This separation should be distinct enough to avoid dependencies between, or interwoven control flow of, main and exception detection code, but not so distinct that the relationship between the two becomes obscured. Separated from these assertions, conventional code can be made simpler to understand and easier to maintain. Assertions should also serve to actively document program semantics, affording the user with a greater appreciation of the limitations of a type. Users should be encouraged to use assertions in their own objects as widely as possible and the mechanism should be flexible enough to support assertion of any aspect of program behaviour or state.

Detection concerns how and when exception assertions are evaluated and the conditions under which the search for a handler begins. Detection should be as immediate as possible, such that the search for a handler is engaged before the original problem has *avalanched* beyond repair. The detection of software exceptions should adopt the same form as that of hardware exceptions to allow unification of the two schemes and a reduction in language complexity. Assertion evaluation should be automatic and follow a strict (regimented) convention, establishing clear responsibilities

[1] A object fails silently if it violates it specification in a way that cannot easily be observed at the time of failure.

between objects, to prevent the user from omitting or making redundant checks. The generation of signals should be automatic to prevent user error or undisciplined signal 'stifling'.

Linkage refers to the language mechanisms employed by the user to map exception violations onto handlers, i.e. that which directs the search for a handler once a violation is detected. It involves the specification of exception-handler pairs (*mappings*) and the association of such mappings with specific execution contexts. Linkage should be as flexible as possible while retaining readability and run-time correctness. That is, it should be relatively easy to read from the source text which handler will be invoked from a particular context after a given violation. Furthermore, a compiler should be capable of deducing that a handler exists for every conceivable exception violation. The granularity of mapping association should be the minimum needed to maximize flexibility, without compromising the underlying language or introducing excessive verbosity.

Handling mechanisms concern the scheduling of exception handlers and their control and data flow. This mechanism should pass all the required information about the circumstances of the exception into the scope of the handler, without violating encapsulation. In addition, if the context of the failing method offers inadequate scope for information or control to decide and implement a final handler response, then the exception should be propagated—without violating encapsulation. A wide range of disciplined, yet flexible control flow models should be available to handler implementors to facilitate a choice of explicit exception responses: from graceful failure to algorithmic redundancy. The former is most commonly used (see Section 2.3.8) and should be well supported. Again the issue of physical separation is important. All handlers should be modular. Each should be distinct enough from the error that it addresses, so that it may be reused in a different (but analogous) situation, but intimate enough to address the signalled problem. Separation has two advantages. Firstly, the textual demarcation often greatly simplifies method code, which makes it easier to understand and review. Because exceptional cases, by definition, occur infrequently, little semantic knowledge is lost by relocating the clutter of exception handling from the main method body. Secondly, the scope in which an anomaly is detected is frequently not the most effective one in which to handle it and separation can make handlers more disciplined by restricting this scope. In an environment with inheritance, this discipline helps to promote reusability.

The detailed design considerations of EPSs in an exception handling environment were considered in Section 4.3.9. Whilst they are fundamentally the same here, the Solve language will have some influence on the demands made on the formalism. Furthermore, some knowledge of Solve is important to understand the significance of the features proposed later. Consequently, we include here a brief discussion of the salient features of the language.

5.3 The Solve Language

5.3.1 Goals and Characteristics

Solve is one of the languages of CoSIDE (C++ or Solve Interactive Development Environment) and was developed at University College London as part of the ESPRIT SPAN project [RSHWW88, RWW88a, RWW88b]. Solve is a parallel, object-oriented language designed for use with parallel, multi-processor systems. It supports fully active (autonomous) objects with concurrently executing methods. Furthermore, Solve provides a message passing subsystem which allows applications running outside its run-time system, perhaps in different languages or systems, to inter-communicate. In general, object inter-communication can be achieved synchronously, asynchronously or with futures. Solve supports type conformance, parameterized types, and multiple inheritance with path linearization to resolve conflicts.

Work on the Solve language aimed to determine the architectural features necessary to support object oriented programming in a parallel system and the feasibility of using a message passing subsystem to integrate diverse software elements. Solve was designed: to harness the isomorphism between objects and processes, and message passing and inter-process communication; to support high level manipulation of complex architectures (with minimal dependencies on those architectures); and to enable the definition of an application, in terms of objects, to implicitly identify application processes that can be executed in parallel.

The language is very flexible and many aspects of it can be fundamentally altered by the user. For example, the mechanisms controlling inheritance lookup, method scheduling, and selector-method binding are essentially methods of the type object and can be replaced by the user. This, coupled with the fact that the Solve language was developed locally and is still evolving, makes it a good substrate on which to determine the efficacy of EPSs.

5.3.2 Solve Objects

In the Solve run-time system, objects are autonomous entities which manage zero or more method processes. These processes may access and alter host object state or communicate with other objects by message passing. Method processes are initiated by incoming message traffic received at the communications interface (see Figure 5.1). Each object encapsulates a state, which consists of an environment which binds names to variables. These bindings collectively denote the object's value; each variable itself references an object (see Figure 5.2).

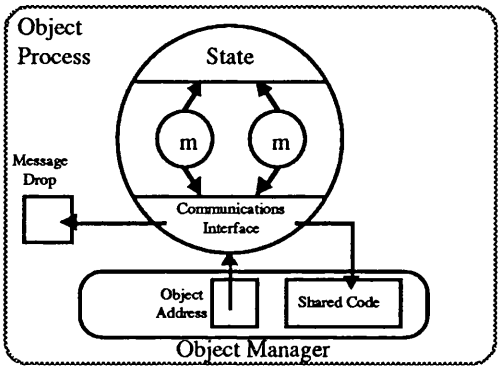

Figure 5.1: A Solve Object at Run-Time

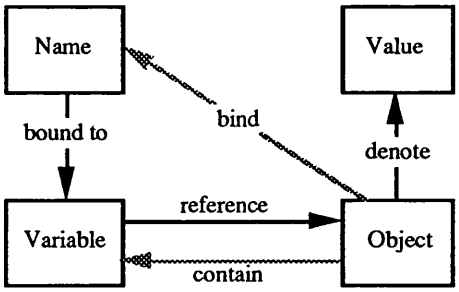

Figure 5.2: The Relationship between Objects, Variables and Value in Solve

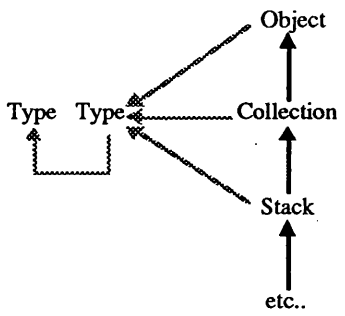

Figure 5.3: The Solve Type Inheritance Hierarchy and Type *Type*

All objects are characterized by their type, which is denoted by the eponymous object variable *type*. If an object's *type* variable references a type object x, that object is said to be an instance of x. Similarly, the methods for performing name binding and monitored access to variables are referenced by the object variables *lookup* and *monitor*. An object's type defines which messages it understands and how it reacts on receiving them. Types may be composed from other types by inclusion (*is-part-of* relationship) or inheritance (*is-a* relationship).

The type inheritance hierarchy is a very important part of Solve. It is a directed acyclic graph (DAG) with concrete types near the leaves and abstract types near the root. Concrete types have instances that are directly manipulated by applications to meet their ends, whereas abstract types cannot be instantiated and serve only to represent the commonality or default behaviour of their subtypes. Types are themselves instances of type *type* (see Figure 5.3) and the latter is an instance of itself.

5.3.3 Solve Type Objects

Like other objects, types have variables, two of the most important of which are *Signature* and *Implementation* (see Figure 5.4). The first declares the messages to which the type object and its instances can respond (the protocol), and the second defines the implementation of the internal state, type methods and instance methods. Specifically: the signature tells us the name of the type, the types it inherits from, the messages it understands and the number and types of their arguments; the Implementation tells us the type and initial values of the state instance variables and the methods of the type and its instances. Signatures are the public part of a type, they *advertise* its facilities to all potential clients and the compiler uses the information therein to perform type checking and incremental compilability. The Implementation of types is hidden from their clients to promote encapsulation and locality.

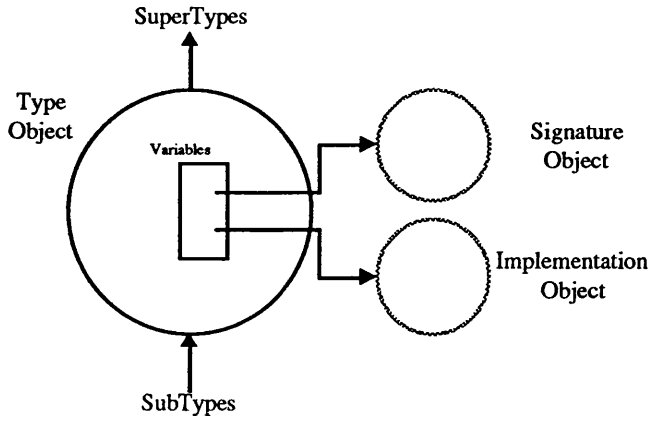

Figure 5.4: Solve Type Objects

All activity in Solve is initiated by message passing which actuates the execution of methods. Most methods lead to further message passing and the cycle is eventually broken by *primitive methods* which are single purpose operations on the underlying virtual machine and initiate no further message passing activity. Primitive methods are the *instruction set* of the Solve virtual machine that enhance its portability. Typically they include atomic operations like addition or reading a character from a data stream.

All non-primitive methods are defined using a small, orthogonal group of constructs: assignment, loops, if-then-else, executable closures[2] (i.e. blocks), sequences and message passing. The *method simplicity* rule of Solve dictates that each method has one goal and is usually less that a page of text, although this is not enforced by the compiler.

Assignment, in Solve, is achieved by altering a variable's bindings rather than the values of objects to which those bindings refer. This, combined with the *commit* semantics of assignment, which mean that the effects of an assignment are invisible until it is complete (committed)—making it appear atomic, helps to ensure object consistency in a parallel environment.

5.3.4 The Addition of EPS

We have augmented the Solve language to support in-source specification through the EPS formalism as part of an orthogonal, general-purpose assertion/exception mechanism. The addition enables Solve to express behavioural semantics as part of the signature of each type. In addition, we have added the more traditional concepts of precondition, postcondition, domain and in-line

[2]The methods of a type object are variables which reference executable closures.

assertion to support a regimented exception handling system (see Section 2.3.5). The result is a new variant of Solve, referred to locally as "Solve III". The new features of Solve III (henceforth abbreviated to Solve) are optional, the compiler is capable of processing traditional Solve code. The extra features have been designed to have a minimal effect on the underlying computational model on which Solve is based and *no* effect on the semantics of existing constructs or operators (in accordance with the requirements of Section 5.2.2).

5.4 Exception Handling in Solve Using EPSs

5.4.1 Philosophy

In Solve, the correctness of a type is defined in terms of labeled assertions, both traditional state based assertions, and as behavioural assertions using EPSs. Assertions relate closely to axioms in ADT theory—they are rules that document source code. Exceptions are internal notifications, or signals, that occur when violations of these assertions are detected. They allow detection of a problem to be broadcast to contexts able to handle it. Solve assertions are written with the primary goal of forcing any program error (that which prevents software meeting its specification) to cause an exception before it manifests itself as a failure. Handlers serve to restore order, maintain correctness, help debugging and support fault tolerance by bringing a program to a point of graceful failure, or if possible, to a point where it once again conforms to its specification.

Each object is responsible for handling its own exceptions and, if they are unable to effect a complete recovery, notifying their clients of their failure in a uniform and disciplined way. The Solve exception handling mechanism is designed to ensure that a method suffering an exception either: repairs itself, achieves its goal and then passes all the checks it previously failed; or reports a failure to its client. No other courses of action meet the correctness requirement of Section 5.2.2.

In Solve, assertions and exceptions are not objects. Assertions are rules that support the correctness of objects and exceptions are messages which report problems. Both help to define an object, but, like a type's name or a message, they encapsulate no methods or internal state of their own and thus are not objects in their own right. They cannot stand alone as independent entities or be modified at run-time. Consequently, we find ourselves unable to support the popular argument, adopted by many other languages (see Section 2.3.5), that either assertions or exceptions are objects. We feel that such an view is driven more by convenience of implementation (e.g. of features like exception parameterization) than any design based reasoning.

Exception handling is supported in Solve using new language keywords to aid separation, readability and compile time checking. We feel the alternative: an exception handling mechanism of similar functionality provided using continuations and no explicit syntactic support (as proposed by [Goo75] and implemented in [Knu87]) is not tenable. The latter approach reduces the readability of exceptions, the ease of separating them from conventional code and their value as 'documentation' because they are couched in the same syntax as the rest of the method. Frequently, new concepts like continuations and non-local goto have to be introduced into the host language to manage the control flow of exception handling. The addition of these concepts can have detrimental effects on the discipline of the language. We believe the introduction of keywords and mechanisms local to exception handling is a far safer technique.

Solve exceptions can be disabled, we consider this as a pragmatic concession rather than one with any formal justification. We view such an activity much as one might view disabling type checking during compilation to speed up the process. The benefits are guaranteed, immediate—but slight—and the drawbacks potentially subtle and costly. Rather than allowing various levels of activation, as other languages do, Solve exceptions are either fully enabled, enabled only at the detection level (all exceptions resulting in reporting and termination) or disabled. We do not believe that selective enabling of assertions of a certain type, a facility offered by some programming environments, is safe or desirable.

5.4.2 Placement

Placement, in Solve, is supported by *signallers* which detect anomalies in objects. Each signaller encapsulates a *requirement* of the object's state or behavioural domain and each has a name. Five types of requirement can be used, four state based and one behavioural. The type of a requirement determines when it is evaluated by the system. Each signaller is, by virtue of its requirement type, associated either with a method of the object (a *method bound* signal) or with object instances (a *instance bound* signal). If any signaller detects that its requirement has been violated, it generates a signal of a type derived from the requirement and bearing the name of the signaller. The requirement types are:

- **Precondition**. Preconditions are method bound state requirements. They consist of an executable closure (block) containing a Solve boolean expression that is evaluated prior to method execution. The only objects in scope within this block are the host object prior to method execution and the arguments of the method invocation. Precondition requirements usually express those conditions which must prevail for method execution to have any

meaning, i.e. they ensure that the method is being applied within its domain. Because preconditions cannot directly access the type implementation, they are not dependent on it and may be located in type signatures without violating encapsulation.

- **Postcondition**. Postconditions are method bound state requirements similar to preconditions. Their closures are evaluated after method execution. The scope of the closure includes the host object, the arguments of method invocation (as they were both before and after the method execution) and the result of the method. Postcondition requirements ensure that the method execution succeeded and produced a result within its range.

- **In-line assertion**. In-line assertions are method bound state requirements similar to preconditions. They are constraints on the ephemeral aspects of state which exist only as an intermediate product of method execution. The scope of in-line assertion closures is that of the method with which they are associated, consequently they are implementation dependent. In-line assertions are used to ensure that the incremental progress of a method is as expected.

- **Domain**. Domains are instance bound state requirements which are evaluated before and after all method executions. They are implementation dependent, Boolean expressions. They express the domain of an object's representation: the constraints on the values held by the object's components that must be satisfied for them to represent a valid instance. Domains may be temporarily violated during the execution of any method, but when the method completes domain requirements must be satisfied (even if other methods are concurrently active). This does not apply after the execution of the method $destroy_{\perp}$ in which case the object state is deallocated.

- **Event pattern specification**. EPSs are used as instance bound behavioural requirements, each of which embodies a legal pattern of behaviour for the instances of the type to which it belongs. The full syntax is as explained in Chapter 4. Solve has no use for action clauses and they have been omitted. EPSs are compiled into parsers, the state of which are reevaluated on the occurrence of each relevant event the object engages in. EPSs have no access to the type implementation and thus they may be placed in the signature of a type.

All signallers, except those having in-line assertion requirements, are physically separated from the implementation of the methods, despite being defined as part of the same object. In accordance with placement design requirements, their affiliations are reflected syntactically without sacrificing their modularity or their ability to be reused directly or through inheritance. In-line assertions must be inserted inside the bodies of methods in order to access intermediate values generated

therein. For this reason, signallers based on in-line assertions are conceptually less elegant than other types and are only used when the requirement cannot be expressed by any other means.

The wealth of requirement types may initially seem over complex, and redundant. EPSs already possess the functionality of preconditions and postconditions (see Chapter 4), but they have been included separately to enhance ease of use[3] and efficiency. Also, the violations of protocol indicated by an EPS failure are generally more severe than the domain and range failures trapped by preconditions and postconditions and therefore merit different handling. Domains may seem nothing more than a method of expressing requirement commonality in the preconditions and postconditions of a method, but this is not the case. They constitute a valuable documentation tool in their own right, are defined in a unique manner and indicate errors of object integrity which may need to be handled in a special way, somewhat different to errors with method applicability or success. The net result of supporting these requirement types is that any manifestation of error can be caught *when it occurs* using signallers (see Section 5.2.2).

Being expressed in terms of the object's encapsulated state, both domain and in-line invariant signallers are part of the implementation of a type. However, other signaller types are part of the signature of a type without violating encapsulation. This has three advantages:

- **The failure modes are 'declared'.** To ensure that each client of an object is aware of its possible run-time failings and that it handles them, one has to declare the signals a type's instances may propagate. In Solve, the place for such declarations is within the signature. By *defining* certain signallers in the signature, this declaration is achieved without further effort.
- **Readability is enhanced.** By making both the declaration and definition of signallers public, the self documentation of the language is improved.
- **Signallers are reusable.** By forcing these requirements to be independent of the representation of the object, they are more abstract and may be reused on types with an analogous protocol.

Naturally, any other propagated signals (which cannot be defined in the signature) also need to be declared in the signature so that clients may expect and handle them. The compiler can only deduce that a handler exists for all run-time eventualities, if each type declares a list of the exceptions it can generate in its signature. This is the *no surprises* rule. The alternative: requiring

[3] Many users will already be familiar with the concept of preconditions from formal texts, or practical use in such systems as Eiffel [Mey88].

no declaration, means that the client is unaware of what exceptions to expect and cannot respond to them with any certainty. Each client must elect to terminate (see Section 5.4.5) all unexpected exceptions (a somewhat limited response) or allow such exceptions to be propagated to its client in the hope that it can handle them. The disadvantages of such implicit propagation are its lack of inherent discipline and the possibility of violating encapsulation [Goo75]. Instead, our approach demands that all exceptions be explicitly propagated (see *blind propagation* in Section 5.4.5). The user intensive nature of this approach may be ameliorated with good development environments, but none the less forces a healthy review of all objects effected by the addition of a new exception.

5.4.3 Detection

In Solve, the detection and signalling of exceptional conditions is done automatically. That is, requirements are checked at a time determined by the run time system and the type of requirement and if the requirements are not upheld, exception signals are asynchronously delivered to the host object. Consequently, the user is only able to choose when in-line assertion requirements are evaluated (during execution of a method), all other requirements are checked in a strict order determined by the run time system. Furthermore, raising of exceptions is implicit. This arrangement has four advantages:

- **Requirements can be more abstract.** Because the evaluation time of most requirements is fixed by the run-time system, the user cannot infuse them with checks that are any more temporally dependent than preconditions and postconditions. This makes requirements more abstract and easier to reuse.
- **Repetitious checking is eliminated.** The responsibility for evaluation and signalling is unambiguously defined. Having specified the requirements once, the user can be sure they are evaluated, and signalled if necessary, at all relevant times. There is no possibility of omitted or redundant evaluation.
- **Exceptions are immediate.** Asynchronous signal delivery allows greater immediacy and eases a merger of software and hardware exception handling mechanisms. Operating system and hardware failure (memory management problems, division by zero, message passing to void objects, etc.) can be considered violations of implicit assertions expressed by the underlying run-time system (i.e. the requirements of primitive methods), rather than the user. It is easy and desirable to manage both with the same asynchronous exception handling mechanism.

- **No explicit raising.** Exception handling is more disciplined because arbitrary exceptions cannot be raised without the violation of an accompanying requirement, as they can in other languages (see Section 2.3.6). More importantly, the strict mapping between failure and exception name is enforced and users cannot inadvertently undermine it with explicit `raise` statements.

The order of requirement evaluation, following a message send from an object a to an object b is:

- active EPSs of a, to ensure the send is behaviourally appropriate;
- active EPSs of b, to ensure the receipt and subsequent lookups are appropriate;
- the domains of b, to ensure the object is valid before execution starts;
- the active EPSs of b, to ensure a method execution is appropriate;
- the preconditions of the selected method, to ensure it is applicable;
- the active EPSs of b and any in-line assertions within the executing method, to ensure execution progress is as expected;
- the postconditions of the method, to ensure execution success;
- the active EPSs of b, to ensure the termination was appropriate; and finally
- the domains of b, to ensure that the method left the object as a valid instance.

The wealth of checks ensures that any error is detected immediately and thus handled promptly. Clearly however, an efficient EPS implementation will be required to avoid a crippling overhead.

The commit semantics of binding used in Solve (see Section 5.3.3) mean that before a method terminates, bindings exist to both the old version of the object (before method execution) and the new version. This facilitates and reduces the overhead of querying and comparing of 'before' and 'after' states in postcondition requirements, as covered in Section 5.4.2.

5.4.4 Linkage

Linkage is, mostly, achieved statically in Solve. Each class defines a *Linkmap* which maps signal specifications onto the names of the handlers (i.e. the *shadow methods*, see Section 5.4.5) which

respond to them. Only one mapping per class is defined, and each component maplet can bind a signal (as specified by name, requirement type, host method and class) to one handler (as specified by name). The signal specification can be partial, allowing a flexible n:1 mapping of signals to handlers. This facilitates handler reuse and reduces the need for the *catchall* (or default handler) of other languages. Consequently, all signals in Solve are explicitly bound to handlers.

Using an explicit linkmap aids the distinct separation between methods and handlers and enables the compiler to check that all possible server propagated signals are handled by comparing the range of exceptions declared in the server's signature against the domain of the client's linkmap. The linkmap also enhances readability by textually localizing a detailed account of how an object handles detected problems.

Although each class defines only one linkmap, the association of each mapping is effectively at method granularity because each maplet may specify the method generating the signal. Thus, if an integer is sent two *divide* messages during the course of a method, both of which can fail with a *divideByZero* signal, the linkmap cannot distinguish between the two and maps both onto the same handler. This may seem restrictive, but it enforces an orthogonal and cohesive exception policy and prevents an accumulation of special cases which can over-complicate code. The restriction is partially mitigated by the fact that most methods (should) have simple goals that can be achieved in up to a page of statements (thus reducing the likelihood of multiple usage of the same method) and that, where truly necessary, in-line assertions can be used to deal with special cases.

Within a linkmap, the order of the mappings is important, the first mapping that the exception signal conforms to determines the handler that is used. Usually, specific cases go first and more general ones (with more partially specified signals) follow. The mapping is a (compiler checked) total surjection. The user must explicitly handle all possible exceptions though the linkmap, even if they are local or require propagation only.

5.4.5 Handling

How handlers are represented in Solve is a critical design decision because of its widespread ramifications on the rest of the language. Are handlers a series of commands to the Solve binder and dispatcher to achieve the data and control flow needed to effect recovery or termination, or are they conventional methods? The former is a low-level approach, easy to abuse, potentially difficult to learn, difficult to share with inheritance and a new concept to the Solve language. As methods however, handlers cannot manipulate control flow in the manner which is often required for exception handling and they have unwanted functionality: they can be executed on demand

by message passing. Solve uses *shadow methods*, a compromise which combines the best features of methods and dispatcher/binder command sequences, without adding any fundamentally new, general purpose concepts to the base language.

All exceptions are handled in Solve by the synchronous execution of *shadow methods*. These are local methods[4] which may alter control flow in ways forbidden to conventional methods through the use of *dispatcher primitives*. These primitives allow users to direct messages to the method dispatcher of the object, which can radically alter the control flow of method execution. They may also be used to acquire information from the dispatcher which is not available within the context of the erroneous object, for example the failed exception name, type and source. Each shadow method is defined like any conventional method and follows all of the language rules thereof (e.g. they may be inherited), except that it is: labeled as a shadow (to improve readability); has no signallers; and can only be executed by a instance of a type in response to a signal. No instance may use either its own shadow methods, or those of others, by conventional message passing. These design decisions reduce the impact of handlers on Solve's design (by making them methods), enables full separation of conventional methods and handlers and retains the modularity and reusability of the latter.

Like conventional methods, all shadow methods have full access to the state of the host object. They also have access to the local state of the method and any arguments thereof (if the violated requirement was method bound). Shadow methods usually consist of a set of Solve operations designed to 'clean up' the state of the object and a dispatcher primitive to select the control flow model (see below) to be used thereafter. In accordance with the method simplicity rule defined in Section 5.3.3, handlers should have only one goal and they should be textually small. Any exceptions that occur during the execution of these operations constitutes a *double fault*, which is always handled by immediate termination (see below). Similarly, any misuse of dispatcher primitives causes a double fault.

Solve supports a range of seven handling control flow models, (the first) five of which are based on the Yemini and Berry model [YB85]. Each is invoked as a control dispatcher primitive with one or more arguments. All control dispatcher primitives have an optional argument, *level*, which defines to what extent the user is notified of the exception and if it is recorded. The models are defined below.

- **Terminate.** Execution of the failed method is abandoned. The error is reported by whatever means are possible and then the process that caused it and all of its parents are terminated.

[4] The local methods of an object are only available from other methods of the same object, they can not be externally invoked and are not advertised in the signature.

The handler using this technique has an obligation to ensure that termination is as *graceful* as possible, typically though the performance of acts like closing any open files and relinquishing any allocated resources. True termination is not very satisfactory under normal conditions because it denies the clients of a failing object the chance of recovery or even 'clean up'. It is included as a pragmatic concession, a model to be used *in extremis* when any continued execution could be dangerous (e.g. in panic situations like memory failure). A significant problem here is propagating the termination order though the process hierarchy—especially is distributed systems. Often the process genealogy of a system is subject to alteration without warning, when, for example, a parent process completes before its children. Consequently each process should maintain a 'forwarding address', to which it forwards termination orders, which is updated as its genealogy alters.

- **Exit**. Execution of the failed method is abandoned and a result object is substituted in lieu of that lost. The handler that uses this control model must ensure that it can achieve the purpose of the failing method. The control primitive has one argument, an object which is of the same type as, or a subtype of, the failed method's result. This object is substituted as the result of the failed message send and is subjected to that method's postconditions. Violation of these conditions causes a double fault.

- **Resume**. Execution of the method is continued after the point of failure. The handler must ensure that the source of the requirement violation is removed before such resumption is attempted. Once handler execution is complete, the requirement which failed is re-evaluated and if it is still violated a double fault occurs. Because of this re-evaluation, this model cannot be used to circumnavigate signallers in Solve, unlike some other systems (see *Lack of Discipline* in Section 2.3.9).

- **Retry**. Execution of the entire method is abandoned and the same method is called again. The handler using this technique must deduce why the method failed and alter conditions to ensure that a retry will be successful. Solve has a dispatcher primitive for determining how many retries have been attempted. Upper bounds on this retry count and its growth rate ensure that recursive exception-retry loops are avoided; such errors are converted to double faults.

- **Delegate**. Execution of the entire method is abandoned. An alternative method (the selector of which is an argument of this primitive) is scheduled. This primitive offers direct support for algorithmic redundancy. It is incumbent on the handler to ensure the delegate can succeed. To be successful, the delegate must meet its own requirements *and* the postconditions of the failed method.

- **Propagate**. Execution of the entire method is abandoned and a signal is propagated to the client. If the failing object has no client, propagation acts like termination. If the original failure was local, the type of the new signal is that of the violated requirement, otherwise the signal is of the type *in-line assertion* (to reflect the implicit in-line assertion of all methods that their servers meet their requirements). It is important that a signal propagated from a server failure (as opposed to a local failure) should have its name changed, the new name is supplied as an argument to the propagate primitive. Propagating signals without name change (*blind propagation*) is likely to violate encapsulation (objects may become dependent on the labels originating from the implementations of others), but its prevention can not be enforced because it is useful to propagate generic hardware faults (e.g. 'disk-fault') without change. One solution is to omit exception names, but this severely limits the flexibility of the exception mechanism and its self documenting power. Alternatively, one could omit propagation, or limit it to one invocation level, but this is very restrictive (limiting the power of recovery) and compromises orthogonality. Instead, we ensure that all signal propagation is explicit and, where possible, that they are re-propagated under a new name better reflecting the failure at the level of abstraction of the propagating object (see Figure 5.5). This explicit propagation preserves encapsulation, but can be overridden in the case of generic hardware exceptions by re-propagation under the *same* name. Naturally, a perverse user can easily fool this mechanism. The new names may help self documentation and certainly do not curtail the flexibility of the system as the other possibilities do. Explicit propagation means that any new exceptions that are introduced might require extensive class editing—but this is an advantage [Goo75], as it ensures that the user cannot do such a thing lightly without considering intervening objects. The programming environment performs the task of decorating the object signatures with all the signal declarations needed. The only disadvantage is that objects high in the compositional hierarchy may be decorated with hundreds of exceptions—this can be avoided by using linkmaps to merge exceptions.

- **Debug**. The errant method is suspended. Its process is attached to a freshly spawned debugging process. The handler has no obligations when using this primitive. However, such a handling technique is only acceptable during development (as a means of gathering information on bugs) and a more responsible model should be substituted before the software concerned is released. The run-time context yielded by the debugger depends on the type of the violated requirement, these are:

  - **For Preconditions**: client of failed message just after aborted message send;
  - **For Postconditions**: server that failed, at very end of execution, before return to the client;

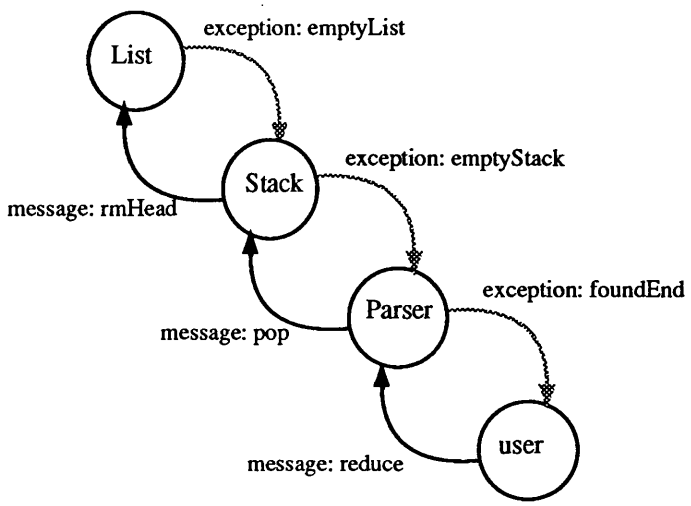

Figure 5.5: Explicit Propagation Showing the Changing Abstraction of Exception Names

- **For In-line Assertions**: server that failed, just after evaluation of the failed assertion;
- **For Domains**: as precondition or postcondition; and
- **For EPSs**: the context issuing the unexpected event.

This flexibility and range of control flow models has two potential prices: program discipline and orthogonality. Program discipline can be compromised if control flow models are used to circumnavigate the exception handling mechanism by resuming execution without reporting or reacting to the anomaly. This is avoided to some extent in Solve by the forced re-evaluation of requirements after resumption and the high level of abstraction of the user's influence on control flow. Some systems offer greater flexibility than Solve, by allowing the user to define their own control flow models (see Section 2.3.1, particularly[Don90, DPW91]) using the host language rather than high level primitives. Typically, such systems allow total user control at the level of stack unwinding and do not force requirement re-evaluation—potentially leading to flexible but unreadable programs with anarchic control flow. Our compromise here is to offer many models, but *not* the facility to design one's own.

The orthogonality of our mechanism is compromised because all of the models cannot be used with all of the signal types and the semantics of a model may vary with signal type (see Table 5.1 for examples). Clearly, behavioural flaws such as those indicated by a violated event pattern specification cannot successfully be handled by the exit or resume models because they do not cancel the occurrence of the offending event. Table 5.1 depicts this and other special cases.

Model	*Precondition*	*Postcondition*	*Domain*	*In-Line Assertion*	*EPS*
Terminate	✓	✓	$✓_1$	✓	$✓_1$
Exit	✓	✓	✓	✓	—$_2$
Resume	✓	✓	✓	✓	—$_3$
Retry	$✓_4$	$✓_4$	$✓_4$	$✓_4$	—$_5$
Delegate	✓	✓	✓	✓	—$_5$
Propagate	$✓_D$	$✓_D$	$✓_D$	$✓_D$	$✓_D$
Debug	✓	✓	✓	✓	✓

KEY:
A tick (✓) denotes that the model can be used with the requirement type, a dash (—) otherwise.

- **1.** The exact semantics of the termination model for parallel object oriented systems will differ if the violated requirement type is instance rather than method bound. If a method fails only the thread which entered the failing method is terminated. However, if an instance bound requirement fails, an object's integrity is impeached and all threads within an object must be terminated.
- **2.** Clearly, to enable recovery from such an exit, all active event parsers would need to be flushed of the events back to and including the execution of the aborted method. If a process hierarchy log is maintained this is feasible, but potentially time consuming and necessitating the locking of the parser's event streams to facilitate alteration. Recovery from an error indicated by an EPS is difficult because the precise cause cannot be communicated, many EPSs have multiple failure modes and to expect a single handler to cope with all of them is dangerous. In summary, the failure of an EPS usually indicates such a sever problem that recovery of any sort is ill advised.
- **3.** Resumption of a behaviourally anomalous method is potentially dangerous for the reasons given for exit above.
- **4.** The number and frequency of retries is recorded as a means of preventing recursion. A dispatcher primitive is available to yield the number of retry attempts made of the currently failing method.
- **5.** Clearly, for a local exception, retrying a method that failed due to a behavioural violation will result in the same violation, since the same method is being executed. As 4, no handler can obtain sufficient information to 'repair' the failed EPS.
- **D.** This is the user's default behaviour.

Table 5.1: Inter-usage of Signaller Type and Control Model

Because both linkmaps and shadow methods are subject to inheritance, users may benefit from the enhanced rigour of exception handling in their own types without having to fully implement it themselves. The default handler (that belonging to the type *Object* at the root of Solve's type hierarchy) handles all exceptions (its linkmap is total) by termination.

5.4.6 Parameterization

The perceived need for parameterized exceptions has had a fundamental effect on many exception handling systems and most of them provide the facility (see Section 2.3.5). It is used in two ways: as a means of conveying certain aspects of the failed object state to the handling closure, so that a handling policy may be altered by, and alter, it; and as a technique for generalizing a family of similar handlers into one. In Solve, parameterization is less useful for a number of reasons: because all shadow method handlers are associated with the failed object, they already have access to its state and via the dispatcher object the handler can determine the exception type, name, method, location and the number of recovery tries made; and access to further elements of remote state is denied to enhance the independence of the handlers on the remote objects. Parameterization can be used with propagation to violate encapsulation by transferring a fragment of object state outside its enclosing scope—a highly undesirable property.

Solve does not support the concept of parameterized generic handler, chiefly because we consider it to be counter-productive. It may reduce the number of handlers, but it increases the complexity of each one (violating the handler simplicity rule) and supports the generation of 'all-case' handlers which are prone to error. We feel this policy, in enhancing handler simplicity, more than compensates for any loss of flexibility it might cause.

5.5 Syntax

In this section, we consider the exception handling syntax of the Solve language with reference to the issues covered above.

Solve retains the same type definition constructs as that of the original language. This structure is designed to enforce the separation of type specification from details of its implementation. Also, it is used to express which methods the instance's clients have access to and which are 'private' (local). Because the issues of specification and client visibility are orthogonal, the addition of in-source behavioural specification to Solve is not as straightforward as it otherwise might be. The behavioural semantics of a type can not always be expressed solely in terms of operations to which clients have access and yet, in Solve, they must be to avoid violating signature encapsulation.

5.5.1 Signature Signal Definitions and Declarations

All Solve types declare a signature which specifies the operations its clients may request of it and its instances and what type arguments such operations expect and yield. This idea is extended in Solve to include a declaration of the name and the type of all exceptions which it may propagate to clients—its *exported signals*. This allows the compiler to check the consistency of all handlers mapped to non-local signals. In cases where requirements have no dependence on the implementation of a type, these declarations may also constitute definitions.

Syntactically, all method bound signals with implementation details are merely declared thus[5]:

```
<Method Declaration>
[
   generates in-line assertion <Name>
]
```

Method bound signal declarations are attached to method declarations. In cases where these signals are of type *precondition* or *postcondition* (and therefore have no implementation dependence) the entire signaller definition is included in the declaration. These annotations inform all clients that the method with which it is associated has the potential to generate the named signal. The syntax is:

```
<Method Declaration>
[
generates
   { signal <Method Bound State Requirement Type> <Name>
   '[' <Solve Expression> ']' }1+
]
```

In a postcondition state requirement, the priming mechanism is used to determine how a certain facet of state changed during the execution of a method. For example, a postcondition requirement to ensure that the associated method increases the value of the instance variable *height*, might be expressed as the Solve expression:

```
(self<--height()) <-- le(self'<--height())
```

Here `self'` refers to the host object after execution. Naturally, priming has no significance in precondition requirements.

All exported signals which are not method bound are declared after the *TypeOperations* section of the signature. For domain signals, which may only be declared, the syntax is:

[5]Note: the outer square brackets here denote that the construct is optional. Square brackets that appear as symbols of the syntax are quoted, viz: `'['`.

```
[ DomainSection
   { signal <Name> }0+ ]
```

Event pattern specifications express the 'temporal protocol' of a type; its behavioural specification. This protocol is defined as a list of event pattern specifications, each of which encapsulates a 'legal' pattern of behaviour for instances of that type. The full syntax of EPSs is described in Chapter 4, and the subset of this syntax used in Solve is depicted below:

```
[ TemporalProtocolSection
   { <Name>
        satisfies <Relevant Trace>
        inwhich <Specification>
        [ iff <Constraint> ]
   }0+
]
```

5.5.2 Implementation Signals

Within its implementation, each Solve type defines the internal representation of its instances and all the operations permitted on that state. The set of defined methods is a superset of those declared in the signature, the extra methods are 'local' in that they are only available to other methods of the host type. In addition, it is incumbent on the type's implementation to define its domain signallers, in-line assertion signallers, linkmap and shadow methods. The state requirements are implemented as Solve expressions of type *Boolean*. Each type of state requirement will differ by virtue of the differences in scope visible from the various signallers (see Section 5.4.3).

In-Line Assertion (ILAs) signallers are expressed *in-situ* as part of the method code. This is the only available technique of facilitating assertions on the incremental progress of a method. The boolean expression forming the requirement has the broadest scope of visibility of any state requirement, able to see all local method variables in addition to the host's instance variables. ILAs are expressed as **ensure** statements embedded in methods. Methods have the amended syntax:

```
<Solve Method Header>
'['
   { <Solve Statement> |
   Ensure <ILA State Requirement>
   } 0+
']'
```

The *domainsection* of a Solve implementation is used to express the state invariants of a type. The section is the last item of the implementation in which it appears, making review of this most critical set of state requirements easier. The syntax of this section is:

```
[ DomainSection
   { <Name> '[' <Invariant State Requirement> ']' }0+
]
```

5.5.3 Linksection

Linkmaps are expressed in the *linksection* of Solve implementations. This is placed between the definition of the type methods and those of the shadow methods. The *linksection* is a list of **handles** clauses, each of which maps a list of signals on to the shadow that is designed to react to them. Any signal which can occur and which does not appear in this map causes a compilation error. The syntax of the section is:

```
[ LinkSection
   { handles <SignalSpec>{, <SignalSpec>}0+ with <Handler Name> }0+
]
```

The order of the handles clauses is significant. The handler scheduled in response to a signal is that denoted by the first **handles** clause with a signal specification (SignalSpec) to which that signal conforms. Signal specifications may be partial and have the syntax:

```
<Signal Type>::<Signal Name> [ @ <Type Name>::<Method Name> [ during <Method Name> ] ]
```

The optional '@' clause represents the origin of non-local signals, if it is omitted only local (non propagated) signals will match the specification. The optional '**during**' clause is used to specify which method is executing when a propagated signal occurs, it has no meaning for local signals. The wildcard character '*' may replace any of the qualifiers above yielding a partial specification.

5.5.4 Handlersection

The syntax of the *handlersection*, being similar syntactically to the *instancesection* in which types define their methods, needs little explanation. It is situated between the linksection and the domainsection of a Solve type implementation. It consists of a list of local shadow methods, syntactically distinct from conventional methods due only to the presence of dispatcher primitives (see below).

5.5.5 Dispatcher Primitives

Traditional Solve defines the message director *super* to enable dynamic channeling of messages to the supertype of the host object. In our proposed version we introduce the director *dispatcher* to allow messages to be sent to the run-time method dispatcher to alter its behaviour. The set of messages understood by the dispatcher are known collectively as the dispatcher primitives. An invocation of such a primitive has the syntax:

```
self <-- dispatcher~<Primitive Name>
```

Each primitive is designed to alter the flow of control, once the handling shadow method has recovered an object or prepared it for graceful failure. Their full semantics are listed in Section 5.4.5.

5.5.6 Example

As an example of this syntax, a commented Solve type can be found in Appendix C.

5.6 Limitations

Despite the strengths of the behavioural exception handling mechanism described here, there are still minor weaknesses in the design which have yet to be resolved. For example, the state requirements of EPSs, domains, preconditions and postconditions, which may use message passing, should invoke only *accessor* or *creator* methods. If they contain *transformer* methods, the semantics of the assertion in which they are involved will be greatly complicated by side-effects. This could be prevented if transformer methods were identified by the language, then the use of such methods in state expressions could be banned. This is not done in Solve, although some dialects of C++ have such an annotation. The overt identification of transforming and non-transforming methods could also be used to enhance readability and determine which methods are applicable to designated *constant* instances. The problem is, who identifies transformer methods? If it is incumbent on the user to do so she may make mistakes. However, automatic classification may not be possible due to the problems of maintaining the dependency graph needed to identify transformers, cycles in that graph and missing implementations (types without implementations are legal in Solve, provided that execution is not attempted). Another solution is to checkpoint the object before a state requirement is evaluated, and *always* roll back. This prevents any side effects but has a considerably high overhead.

As discussed in Section 5.4.4, the linkmap only allows mappings to be defined with method granularity. The syntax does not allow the user to associate a signal-handler mapping with a particular statement in a method. This limits the power of association provided by the mechanism, but ensures consistent use of handlers within a method. Short of providing line number references, association cannot be made more specific while lexical separation is enforced.

To some extent, Solve's separation of the Signature and Implementation construct serves two purposes—it is semantically overloaded. The separation of external functionality from internal functionality has been inexorably bound to the separation between abstraction and implementation. This is correct in that all external services should be advertised in the signature and aspects of the implementation should be private, but the behavioural specification of a type should reference the local methods. The dual rôle of this separation has some advantages, but it does prevent local methods from having any form of exceptions associated with them, limiting the scope of usefulness of exception handlers.

One could argue that the goals of readability are compromised by the mathematical nature of EPSs. Few formally derived temporal algebras are easy to use or read. Other representations of EPSs (e.g. graphical or analogical) may alleviate this.

Currently, there are few ways for EPS to communicate the precise reason for their failures to their handler. This unfortunate limitation has two possible solutions. One could extend the host language to include built in support for events, traces, etc and allow handlers to query parsers directly; or, more pragmatically limit one's use of EPSs to monomodal examples—those that can fail for only one reason.

Chapter 6

Implementing An EPS-Based Exception Handling Mechanism for Solve

6.1 Introduction

The feasibility of EPS can only be ascertained by formulating and coding an implementation. In this section we consider the salient implementation issues of Solve's EPS-based exception handling mechanism. The bulk of Solve's implementation, including the exception handling mechanism discussed in Chapter 5, is *not* exhaustively explained here. This is because it does not constitute original research material by the author and because such details are more fully referenced elsewhere [RSHWW88, RWW88a, RWW88b]. Instead, we focus on the interesting problems encountered during implementation of those components of Solve for which we are responsible, defining only those parts of the Solve implementation that are required to facilitate understanding of these problems.

Implementation efforts were driven by three criteria: efficiency, to permit EPS to be used without curtailing the usefulness of the system; flexibility, to enhance the ease of further change and improvement; and congruency, to minimize the impact on the underlying Solve implementation and to introduce as few new 'special cases' as possible.

The exception handling system introduces four broad capabilities to the Solve language which can be considered and implemented separately. These four steps are also those required to 'port' EPSs to another language. These are: the extension of the language definition while retaining backward compatibility and compilability; instrumentation—automatically forming an event

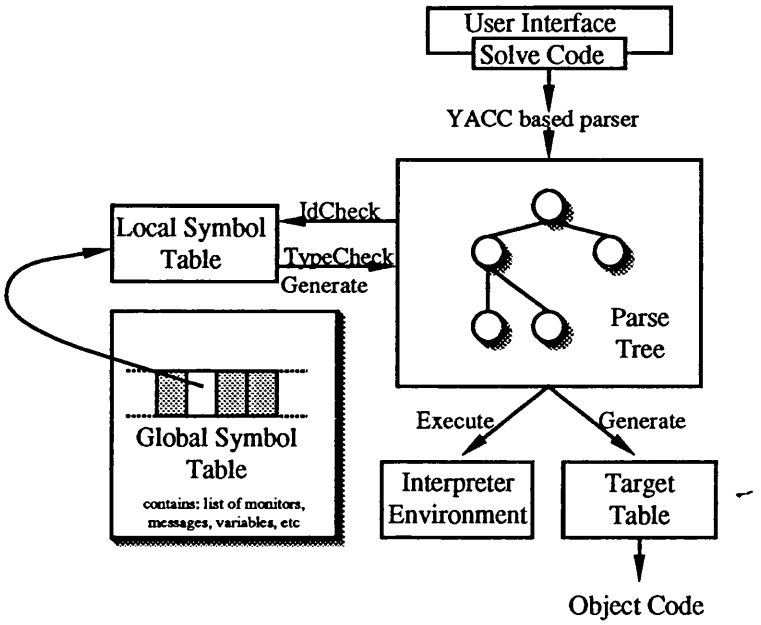

Figure 6.1: The Solve Compiler

stream representing the behaviour of the Solve system; EPS proving—establishing which specifications are violated; and exception handling—supporting the defined control and data flow models (see Section 3.3) in a parallel environment. EPS proving concerns issues pertaining to event stream filtration, pattern matching and constraint proving. Below, we consider each of these points in turn, examining the major problems of each and (where appropriate) the solutions, before considering the status and limitations of the implementation as a whole. This is prefixed by a brief overview of the underlying Solve implementation to provide the reader with sufficient context.

6.2 The Standard Solve Compiler

The Solve compiler/interpreter, as shown in Figure 6.1, is an environment for creating and manipulating Solve parse trees. Currently it is implemented in C++. Parse trees are acyclic nets, created by a YACC parser from Solve source code, each node of which is an object. Over 27 node types exist, representing both Solve language constructs and more abstract concepts. Most nodes are non-terminal and have other nodes as subcomponents; the node hierarchy of an example type is shown in Figure 6.2. Later, we show how this structure alters as a result of our additions to the compiler. Notice that a type's instance variables (represented as *VariableNodes*) include its methods, which are variables that represent executable closures. At the leaves of the node hierarchy are

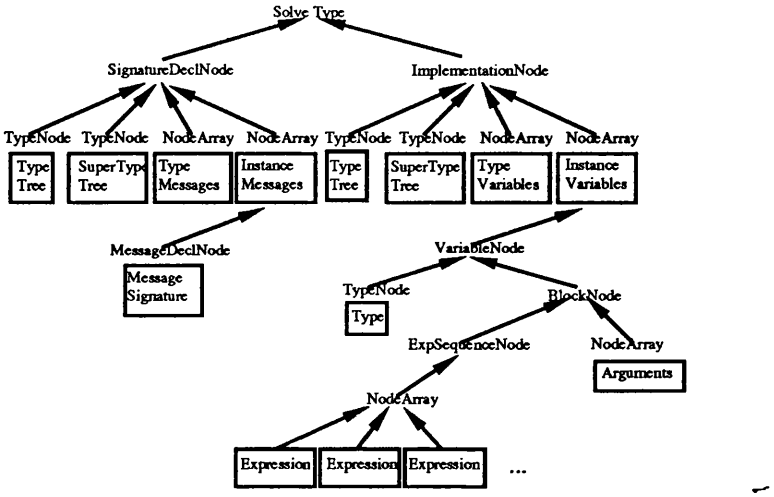

Figure 6.2: Node Structure of Solve Type

the self contained terminal nodes, e.g. these representing fundamental operations (*PrimitiveNode*) and values (*LiteralNode*).

Node types form an inheritance hierarchy (see Figure 6.3). Some nodes in this hierarchy represent abstract constructs like sequences of executable instructions (*ExpSequenceNode*), collections of nodes (*NodeArray*), the root node (*MainNode*) or type implementations (*ImplementationNode*, refer to Section 5.3). Other nodes represent concrete language structures like assignment (*AssignmentNode*), message invocations (*MessageNode*) and lexical closures (*BlockNode*). Later, we outline the new nodes necessitated by our language additions.

All nodes inherit their basic behaviour from the type *ExpressionNode*. This behaviour consists of a standard protocol of five methods:

1. **idcheck** causes a node to generate a local symbol table and check the use of its identifiers against this table;

2. **typecheck** causes a node to type check the language structure it represents and update the local symbol table with type information, the type of the expression represented by the node is also stored;

3. **generate** causes a node to write a code template for the structure it presents, make appropriate substitutions for the target platform and dump the resulting code to a file;

4. **execute** causes a node to interpret itself and update the state of the Solve virtual machine (its interpreter) accordingly; and

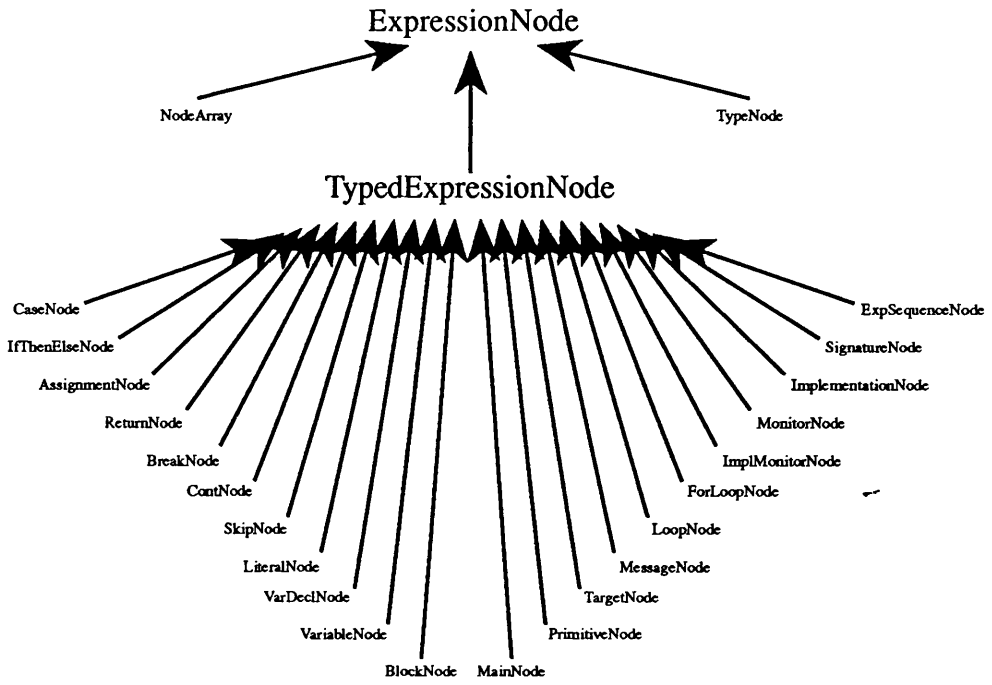

Figure 6.3: Parse Tree Node Type Hierarchy

5. **print** causes a node to produce an indented textual representation of itself to facilitate code browsing.

On receipt of any of these messages, non-terminal nodes are obliged, at the very least, to propagate them to their subnodes. Consequently, to produce a complete textual representation of a program, it is only necessary to send a *print* message to the parse tree root node. The message sequence *idcheck*, *typecheck* and *generate*, when sent to the root node, effects a full compilation (including code generation) of the parse tree.

As an example of this propagation, consider a three line Solve program like that of Figure 6.4. When parsed this forms a node graph resembling the solid features of Figure 6.5, for brevity we have omitted all of the details concerning the AssignmentNode and the second MessageNode.

One can see the effects of interpreting this tree by conducting an in-order traversal of the grey arrows of Figure 6.5. Initially the root node, a MainNode called *mainnode*, is sent the message *execute* (see arrow 1 of Figure 6.5). This precipitates a chain of inter-node message sending. The MainNode propagates this message (2) to its main constituent, the ExpSequenceNode *maintree*, which in turn (3) forwards it to the NodeArray *exps*—representing the array of expressions in the program. The NodeArray propagates the message to each of its children *in turn*, the first

```
Program Simple
    Screen <-- Clear();
    let i <Integer> := 0;
    Screen <-- PrintNl(i)
End
```

Figure 6.4: Solve Example Program

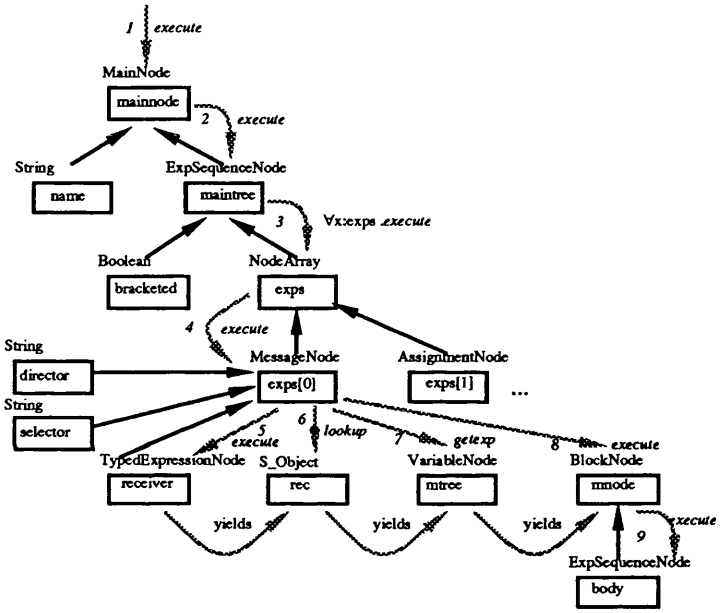

Figure 6.5: Node Interpretation of Program "Simple"

of which is the MessageNode *exps[0]* (4) which represents the expression '`Screen <-- Clear()`'. The MessageNode first evaluates the receiver of the message (by sending it an *execute* message, 5) yielding a Solve interpreter object (*S_Object*[1]) *rec*. It then retrieves the definition of the selected method from the receiver (6), yielding a VariableNode *mtree* (recall that all Solve methods are represented by VariableNodes). Finally, it extracts the executable information from this node (as a BlockNode) and executes it (7, 8). The BlockNode propagates the execution message to its component ExpSequenceNode *body* (9), which further propagates it in a manner analogous to stage 2. The stages of interpretation 2-9 represent a method invocation cycle in Solve and such cycles continue recursively until terminal nodes are encountered. Later, we show how this cycle is altered to achieve effective event instrumentation.

[1] Instances of the C++ class *S_Object* represent all solve objects in the interpreter.

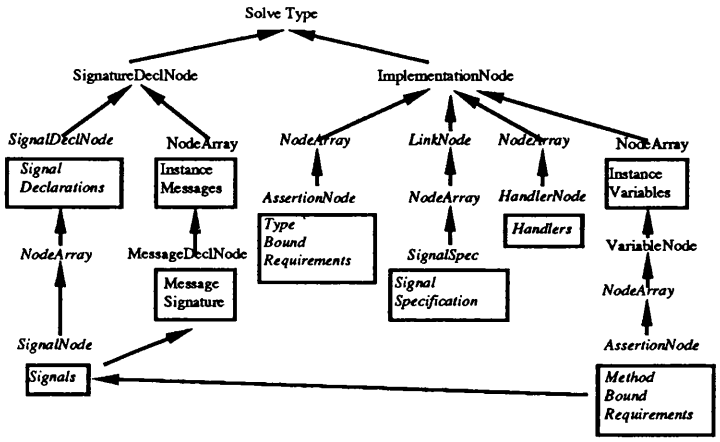

Figure 6.6: New Node Structure of Solve Type

6.3 Language

6.3.1 Syntax

The implementation of Solve's modified syntax (see Section 5.5) was achieved entirely by amendment of the LEX and YACC grammars which produce front end parsers for the parse tree manipulator. Changes were entirely additive in that the new parsers can still parse traditional Solve code. We attempted to add as few new lexical concepts as possible, integrating many of the new keywords and constructs into existing YACC classes.

6.3.2 Exception Signaller Deployment and Detection

Parse trees produced by the new compiler represent types using the additional structure depicted in Figure 6.6. The new nodes names are set in italics in this figure and have a hierarchy as defined in Figure 6.7. *SignalDeclNodes* represent exportable signals declared in a type's signature. Those that are direct components of *SignatureDeclNode* represent type bound signals and the others are method bound. Each is responsible for ensuring that it represents signals of the correct type (in response to a typecheck message) and (in response to an idcheck message) that no name clashes exist before a *SignalRecord*, representing the signaller, is added to the global symbol table *GlobalTable*. SignalNodes within the SignatureDeclNode may represent signal declarations (i.e. inline assertions) or full signaller definitions (i.e. preconditions, which use *AssertionNodes* to represent Solve expressions, or EPSs represented by *EPSNodes*).

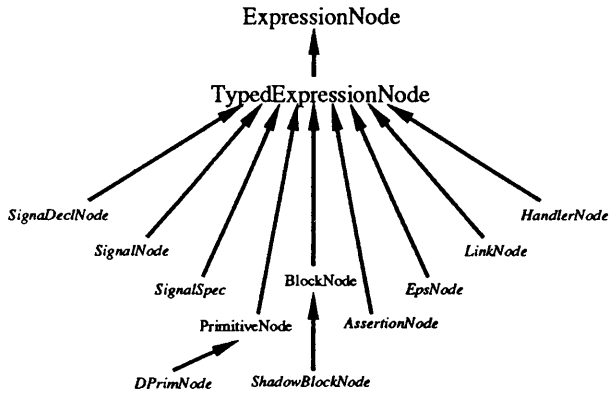

Figure 6.7: New Parse Tree Node Type Hierarchy

AssertionNodes belonging to ImplementationNodes represent type bound requirements (i.e. Domains). At compile time, AssertionNodes ensure that the type of requirement they represent is compatible with the node to which they belong. Furthermore, they ensure their consistency with the declared exported signals of the signature (in the *SignalDeclNodes*).

Those AssertionNodes representing in-line assertions are contained within the executable sequence NodeArray and are executed, like other Solve expression nodes, at stage 3 of the method invocation cycle (see Figure 6.5). Preconditions are checked before stage 2, postconditions on a successful return from stage 2 and domains on both these occasions, this is achieved by instrumenting the Solve binder. Domain evaluation is allowed full access to the environment[2] of the method, whereas the other method bound requirements are not. Postconditions are checked at a time before the method has committed, consequently bindings exist in the solve virtual machine to the host object state both prior and subsequent to method execution. This enables the evaluation environment to be manipulated to support the priming mechanism within postcondition requirements.

AssertionNodes are *BlockNode* variants that must produce a boolean result. It is undesirable for evaluation of these assertions to change the state of the host object in any way, otherwise the evaluation itself could be rendered meaningless (see Section 5.6). Ideally, evaluation of the condition should not be capable of introducing side-effects on the state of the host object, but this restriction is not easy to enforce. Four techniques were examined:

- **Checkpointing**. Snapshot the state of the host prior to evaluation and restore state afterwards;

[2]Environments (or activation contexts) are C++ objects, of the class *Env*, which map names to objects.

- **Preventing assignment**. Prevent any form of assignment within the condition block;
- **'Safe' methods**. Allow AssertionNodes to use only methods guaranteed not to alter the state of any object; and
- **Prevent commit**. Prevent the condition block from committing.

Checkpointing is prohibitively expensive and partially redundant due to the commit semantics of Solve. Preventing assignment, while efficient, is overly restrictive; although, it is worthwhile to generate a compiler warning if a direct assignment is made to a method argument in an assertion. The most complete solution is that of 'safe' methods (see Section 5.6). By allowing the compiler to distinguish between accessor (safe) methods and others—either by user denotation, or automatically by scanning the dependency net for AssignmentNodes or PrimitiveNodes known to alter state—we introduce a language concept that has many uses beyond assertion checking. For example, safe methods could be used as the basis for enforcing constant object instances and they encourage user discipline in separating the accessor and transformer components of object protocols. Safe methods are not new, the C++ language, has 'const' methods which have similar semantics. However, the implementation of such an intrusive concept would have serious ramifications on most aspects of Solve's original design and implementation which are outside of the scope of this research. For example, it restricts the granularity of incremental compilation by increasing inter-method dependencies.

Preventing commit is the least intrusive and most effective of the solutions, especially as Solve forces users to retain a 'functional' approach to the design of transformer methods (i.e. have such methods return a new object rather than modify the receiver). However, it can not prevent side effects to objects like *screen* or *disc* which lack a functional interface. Nor is it able to prevent assignment to method arguments, although the latter can be rendered harmless using argument copy semantics. We use a combination of this method and compiler warning on assignment (see above).

6.3.3 Linkage

In the interpreter, once an exception is detected, a message (representing a signal) is sent to the failing S_Object. This object consults the *LinkNode* belonging to its type object. LinkNodes represent the LinkSection of a type, they consist of a set of mappings of (partial) signal specifications (represented by *SignalSpecNodes*) to handlers (represented by *HandlerNodes*). Individual SignalSpecNodes are able to compare an incoming signal and accept or reject it. When sent a signal, the

LinkNode returns the HandlerNode associated with the first SignalSpecNode to accept this signal. The S_Object then schedules this HandlerNode for execution, as if it were synchronously called from the erroneous context. The environment passed to the handler is that seen by the method and the dispatcher retains information about the exception which is available (only to the handler concerned) through dispatcher primitives (see Chapter 5). This eases the task of resuming normal execution should the handler fulfill its task.

At compile time, each SignalSpecNode establishes the existence of its defining signal name, type, host method, object and designated shadow method. Furthermore, each LinkNode ensures that all non-local signals that are exported by servers of the type are covered by at least one SignalSpecNode. This degree of static analysis is demanding and time consuming, especially if dynamic binding is used in programs (in such cases the type of an instance may be unknown until run time, necessitating an exhaustive compile time search for possible handlers), but it helps to prevent unclaimed signals at run time.

6.3.4 Exception Handling

HandlerNodes are VariableNode variants with executable closures represented by *ShadowBlockNodes* instead of the conventional BlockNodes. ShadowBlockNodes allow both PrimitiveNodes and *DPrimNodes* to feature in their representation, whereas BlockNodes allow only the former. This affords HandlerNodes a superset of the behaviour of VariableNode methods, each DPrimNode representing a dispatcher primitive. The selector lookup algorithm used by the Solve binder ignores all HandlerNodes, thus preventing their invocation or amendment by either external or local agents. Depending on the DPrimNodes used by a particular handler, HandlerNodes are labeled as to which signal types they are fit to handle. For example, handlers containing the `dispatcher_resume()` primitive are not compatible with non-resumable EPS signals. A compilation error will result if an ImplementationNode discovers during typechecking that the type LinkNode shows a mapping between a signal and an incompatible HandlerNode.

Should a handler have the opportunity to and succeed in saving an erroneous context, as we have seen above, continued execution is straightforwardly achieved by allowing the shadow method to return normally. Termination is also easy to implement, although in a parallel environment the timely communication of a process's demise might pose problems. However, the control flow of delegation, exit, exception propagation and retry models all require some form of invocation stack manipulation within the Solve interpreter. This is chiefly because the interpreter is implemented in C++ and not in Solve itself. Stack manipulation can be implemented using distributed unions

or non-local goto. In the former, HandlerNodes and all other Nodes involved in method execution always finish execution normally and return a status object which their client uses to determine how execution should proceed. If each stage of method invocation honours this protocol, it is simple enough for a handler object to (in the propagation example) dictate that execution is to roll back to the context invoking this one and then that an exception is to be delivered. Non local goto performs this stack manipulation directly, without any return. Consequently, the latter is less intrusive, faster, but lacks portability, is undisciplined and unreadable. We have adopted the latter technique, discounting the last two disadvantages on the grounds that Solve users will never directly perceive the control flow used by the interpreter.

The current implementation of Solve uses the C++ functions *setbuf()* and *jmpbuf()* to achieve non-local goto. Figure 6.8 illustrates how each of the control flow models use non-local goto to achieve the context switch after exception handling. Note that if an exception is signalled during exception handling a double fault occurs. At this point a handler built-in to the interpreter is scheduled to handle the error. This handler is not user-defined, as it is in some systems, because its execution is likely to have resulted from human error—to risk further human error by using another user-defined handler is unwise. The double fault handler *notify()* makes a textual report of the exception and gracefully terminates the host process and all dependents.

6.4 Implementing EPS

Providing an efficient and full implementation of Event Pattern Specification is the most demanding aspect of the Solve exception handling system implementation. The task has two principal components which are discussed here: instrumentation, that of producing an event stream and parsing, that of insuring it is as expected.

6.4.1 Instrumentation

The method invocation cycle of Solve is instrumented in ten places to report eight classes of behaviour corresponding to the event alphabet derived in Chapter 3. This instrumentation is diagrammatically represented in Figure 6.9, in which the instrumentation for *send*, *lookup*, *error*, *execute* and *terminate* are represented by the encircled letters **S**, **L**, **E**, **X** and **T** respectively. In addition to reporting the occurrence of particular events, instrumentation provides the event parameters. Common parameters like process identifiers (pids) and time are directly supported by the provision of a logical clock and pid counter[3]. Most other parameters are easily available or

[3] In a distributed system the pid will incorporate the node name.

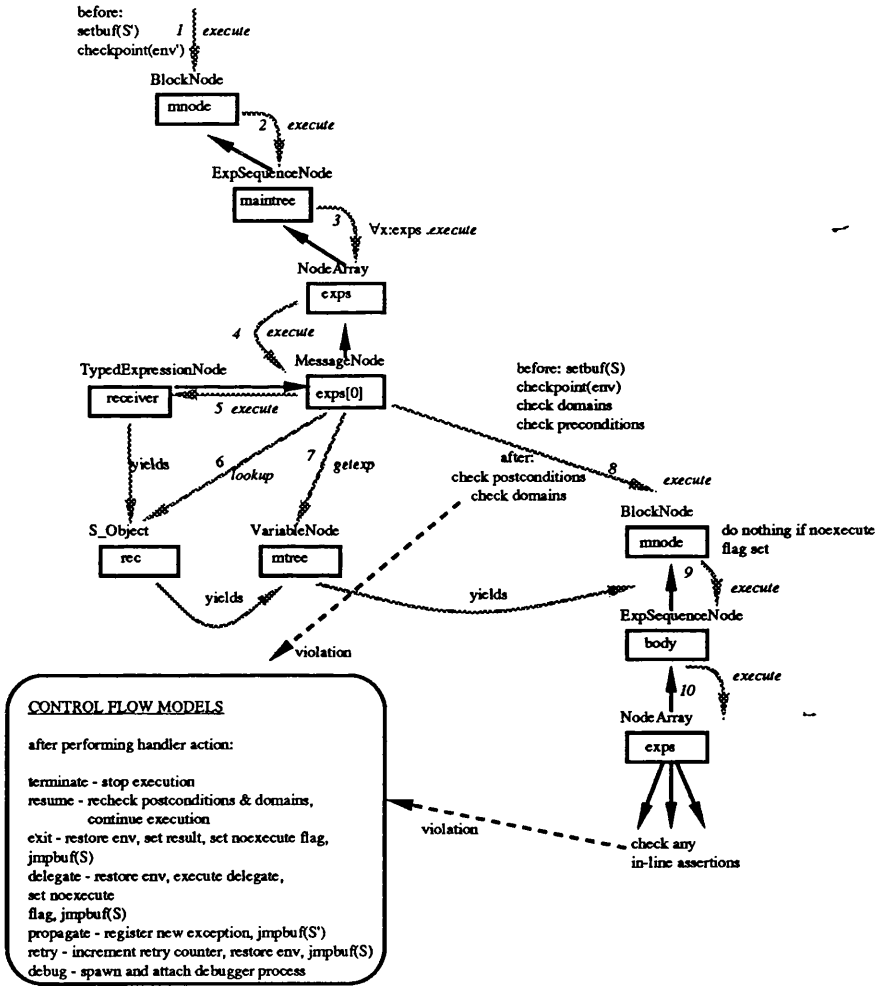

Figure 6.8: Control Flow Model Implementation

calculable at the point of instrumentation. For example, *execute* and *terminate* events are reported as stage 8 of the method cycle starts (see Figure 6.9) and returns respectively[4], all the required parameters are readily available. To capture other event parameters requires cooperation from other Solve subsystems. Consider, for example, the *send* event which requires instrumentation in two places: one (just before stage 5) for incoming messages and another (within *MessageNode*) for outgoing messages. In both cases, the method arguments need to be prematurely evaluated (using *execute*) so that the event report can include their values.

To report *lookup* and *error* events requires instrumentation of the selector lookup and binding algorithm (a method of the C++ S_Object class) with amendments to ensure that both class parameters are available. The constructor and destructor of S_Object are also modified to report *create* and *destroy* events. The required modification is extensive in the latter case due to two main problems: S_Objects are nameless entities (their parent environment determines their name), which makes it difficult for either event to report what object is being affected; and each S_Object may appear in many environments, so one cannot always infer from the execution of the S_Object destructor that an object has been destroyed. To resolve the first issue, S_Objects were modified to retain the *first* name mapped on to them in an environment. This is frequently, but not always, the most significant. The second issue is corrected by using a reference counting algorithm within the constructor and destructor, enabling S_Objects to discern genuine object destructions. Object references (type O, see Section 4.4.3), the most common type of EPS event parameter, were represented in the Solve version of EPS as S_Object names. This feature represents the only implementation specific feature of EPS's design.

Access events require two instrumentations. Both involve modifications to the C++ class *Env*, instances of which represent environments or activation records. Read accesses are reported by modifying the mechanisms for retrieving (non-method) environment bindings and write accesses by similar amendments to those that create new bindings. Again, premature evaluation is required to enable the event report to include the required parameters.

To where should an event stream, generated by instrumentation, be sent? One approach is to centralize event collection at one special object. This *overseer* can then direct debugging and specification based checks, interrupt erroneous objects and forward event streams to other processor nodes for distributed debugging. However this technique has several disadvantages: it necessitates a 'special case' send message primitive to avoid send event recursion; it requires that objects violate encapsulation by revealing their behaviour in intimate detail to others; and it entails

[4] Thus instrumented, *execute* events are reported *after* the method execution environment is created and *terminate* events reported *before* the environment is destroyed, facilitating handler access to it.

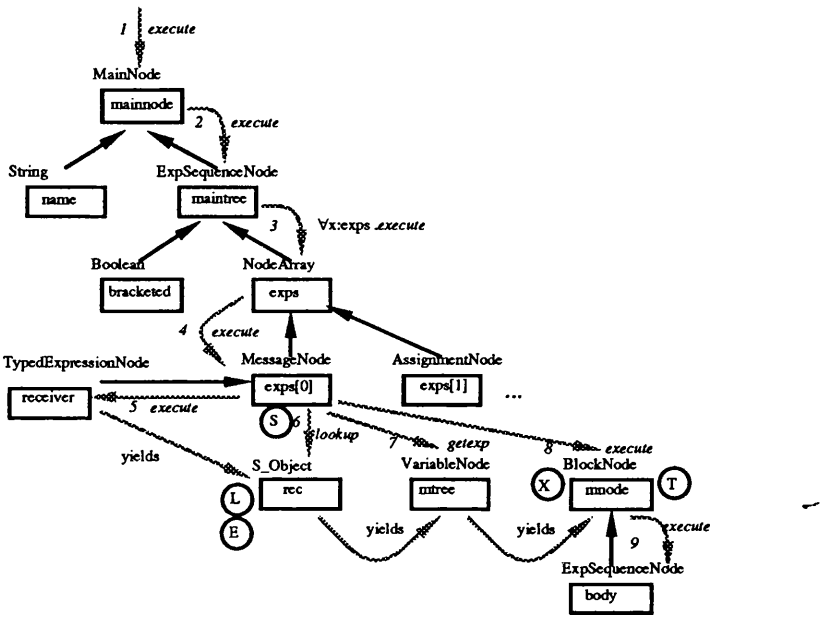

Figure 6.9: Instrumentation of the Solve Parse Tree Nodes

vastly increased communications overhead. A superior approach is to direct all event streams to EPS parsers within the originating object. This approach overcomes all of the previous problems, balances the overhead of EPS very evenly (especially in a system where different objects may reside on different processors) and improves the accuracy of the relative timestamps on the events (which might otherwise suffer from latency).

Once an event is received, all active EPS parsers which have the event in their relevant trace parse it (a process described in Section 6.4.2 below). Any violation causes the parser to execute a handler (which must eventually terminate the thread concerned). Any other active threads within the object at this time are terminated with a generic exception. Naturally, the execution of any handler disables further instrumentation for the thread concerned.

6.4.2 Parsing the Event Stream

The parsing of concurrent event streams involves complexities outside of the realm of traditional parsing techniques. Traditional finite state machines (FSMs) are inadequate for the task because, in order to make a transition, a FSM uses the availability of individual symbols from the alphabet of the machine. Because we are monitoring systems with concurrent event generators, we need a FSM that makes such transitions based on the concurrent availability of *sets* of input symbols. Such a family of automata has been proposed and used successfully by Bates [Bat87b, Bat89, Bat87a].

In addition, Bates' automata have many other properties desirable in this application. His *Basic Shuffle System* (BSS) is capable of delegating the recognition of sub-specifications to subordinate automata by considering them as further symbols in the alphabet of the 'root' machine—thus enabling symbolic delegation and the hierarchical definition and recognition of event-based behaviour. Bates' *Constrained Shuffle System* (CSS) goes further, allowing each transition to be guarded by a set of constraints binding the event parameters, facilitating event filtering, clustering and constraining.

An early version of an EPS parser was partially implemented using a Constrained Shuffle Automaton. Despite offering much of the required functionality, several undesirable properties were discovered:

- CSAs are unable to express constraints on EPS operators or trace variables, only event parameter constraints are available;
- CSAs are ideal for *all or nothing* specifications, but because constraints guide every transition they are poor at behavioural fuzzy matching;
- The *all or nothing* nature of CSAs also impairs their ability to convey how the unspecified portions of a specification were satisfied if a constraint fails;
- The effective efficiency of CSAs is somewhat impaired by the fact that constraints are checked at every transition; and
- CSAs don't reject unexpected events at the first opportunity but retain them in their input register. Therefore, CSAs do not *fail* in response to bad behaviour, rather they just don't succeed.

It is clear that CSAs already offer much of the desired functionality. A modification is required which enables unification between all the variables of a partial specification (both event and operator parameters) and an instance of the specified behaviour. We require a CSA that is not forced to dismiss a partial recognition purely because of a constraint, but that fails immediately if an unexpected event is witnessed. An automaton which is not transition guided by constraints, but which checks a series of constraints subsequent to pattern matching and single-pass unification—allowing more flexible matching and permitting the deferment of computationally expensive checking until after recognition has been achieved. These goals are achieved by using *Unifying Constrained Shuffle Automata* (UCSA), a unification driven variant of Bates' BSS.

6.4.3 Unifying Constrained Shuffle Automata

UCSAs are a modification of Basic Shuffle Automata that perform simple, single-pass symbol unification. Each transition of the automaton, or subordinate automaton, is accompanied by a *unification vector* (of functional *sections*) which dictates how the symbols, to be defined by recognition, change as a result of that transition. When a UCSA reaches its final transition the results of the applications of these vectors is tested against a series of constraints and if they succeed, the behaviour is said to have been recognized. The automaton is defined:

$$UCSA = \Sigma, S, T, A, Z, U, C, F_C, W$$

Where:

Σ	input event alphabet for the UCSA, $\{s_1, s_2, \ldots, s_n\}$
S	set of Shuffle Automata, $\{S_i \mid S_i$ is a Shuffle Automaton$\}$
T	transition sets, $\{t_i \mid t_i \subseteq \Sigma \cup S\}$, $\forall i,j\ t_i \neq t_j$
A	all active automata, $A \subseteq S^*$
Z	vector of unifiable symbols, $Z = \begin{pmatrix} z_1 \\ \vdots \\ z_m \end{pmatrix}$
U	alphabet of unifying sections, $\{I, \mathbf{fix}, = x, +x, -x, /x, *x, \mathbf{concat}\ x, \ldots\}$
C	alphabet of constraining sections, $\{I, < x, \leq x, > x, \geq x, = x, \subset x, \ldots\}$
F_C	the final constraint, $F_C.v_{S_y^x} \mapsto \{True, False\}$
W	the set of m-ary transition vectors, $\{w_i \mid \exists t_i \in T\ .\ w_i \in W\}$, $w_i = \mathbf{vector\ of}\ U$

Naturally Σ is restricted by the relevant trace of the EPS implemented by the UCSA. Here, U is the alphabet of all sections applicable to value variables (see Section 4.4.8) and trace variables (see [Hoa85]), including I the identity section, and **fix** the section that makes a symbol's value permanent. Similarly, C is the alphabet of sections applicable to constraints. The constraint I is always satisfied for it indicates no constraint. The vector Z represents the names of the traces and value wildcards instantiated by the defining EPS. Each element of Z has an associated type z_t (either *trace*, *numerical* or *value*) and an initial value z_s, typically <> for traces, 0 for numerics and '' for strings. The set of transition vectors W describes how the symbol vector (see below) alters for each transition in T.

Each subordinate shuffle automaton has in addition to the above:

Q_{S_i}	set of states for the automaton S_i
m_{S_i}	transition function, $Q_{S_i} \times T \rightarrow (Q'_{S_i} \times W)$
q_{0S_i}	start state for S_i, $q_{0S_i} \in Q_{S_i}$
F_S	set of finishing states for S_i, $F_{S_i} \subset Q_{S_i}$
r_{mapS_i}	input register mapping function for S_i, $(\Sigma \cup S)^* \rightarrow (\Sigma \cup S)^*$
v_{S_i}	symbol state vector, after n transitions $v = \begin{pmatrix} z_{s1}.u_1^1 \circ u_2^1 \circ \cdots \circ u_n^1 \\ \vdots \\ z_{sm}.u_1^m \circ u_2^m \circ \cdots \circ u_n^m \end{pmatrix}$

Here, the symbol state vector v_{S_i} has an element for each name in the unifiable symbol vector, Z. Each of these elements is initialized to the value of the parent automaton's vector (or to its start value z_{sx} if it belongs to the 'root' automaton) and is then changed by the successive application of transition vectors (w_i) associated with every state change (t_i).

Like Bates, we decorate each automaton with a unique identifier, i. New identifiers for sub-automata spawned from i are given a new identifier derived on the parent (i) and the state of parent at the time of spawning (q) using the function $new(i, q)$. The parent of an automaton, i, is denoted $parent(i)$. Unlike Bates' CSA, the transition function m yields a cartesian product of the next state *and* the next applicable transition vector to express the symbol changes resulting from the state change. The functions $\mathcal{Q}$ and $\mathcal{W}$ are used to respectively isolate these two components.

The algorithm for parsing is as follows.

1. The 'root' machine is preset and the symbol state vector is initialized. Any subordinate automata needed to leave the initial state are started. Note that $R_{S_k^i}$ is the input register of automaton i of type k.

$$
\begin{aligned}
&q_{S_0^0} \leftarrow q_0 \\
&R_{S_0^0} \leftarrow \{\} \\
&A \leftarrow \{S_0^0\} \\
&v_{S_0^0} \leftarrow \begin{pmatrix} z_{s1} \\ \vdots \\ z_{sm} \end{pmatrix} \\
&\textbf{forall } j, l : m(q_0, t_j) \neq \perp \textbf{ and } S_l \in t_j \\
&\quad A \leftarrow A \cup \{S_l^{new(0,q_0)}\} \\
&\quad q_{S_l^{new(0,q_0)}} \leftarrow q_{0S_l} \\
&\quad R_{S_l^{new(0,q_0)}} \leftarrow \{\} \\
&\quad v_{S_i} \leftarrow v
\end{aligned}
$$

2. As the symbol generators produce symbols $\{s_1, s_2, \ldots, s_n\}$ use a symbol distribution function to determine which symbols to add to their input registers and which to reject.

input $\{s_1, s_2, \ldots, s_n\}$
forsome $i, j : S^i \in A,\ s_j \in \{s_1, s_2, \ldots, s_n\}$
  if $\exists i\ s_j \in t_i$ **and** $m(q_{S_k^i}, t_j) \neq \perp$
    then $R_{S_k^i} \leftarrow R_{S_k^i} \cup \{s_j\}$
    else reject

3. When the input register of a shuffle automaton in the active set contains one of the outgoing transition sets for the current state of the automaton, finite state control for the automaton makes the transition, altering the input register and symbol vector accordingly. Any new automata that are required are started up.

if $\exists i, j : m(q_{S_k^i}, t_j) \neq \perp$ **and** $t_j \subseteq R_{S_k^i}$
then
  $q_{S_k^i} \leftarrow Q(m(q_{S_k^i}, t_j))$
  $R_{S_k^i} \leftarrow r_{map}(R_{S_k^i})$
  $v_{S_k^i} \leftarrow v_{S_k^i}.\mathcal{W}(m(q_{S_k^i}, t_j))$
  forall $j, l : m(q_{S_k^i}, t_j) \neq \perp$ **and** $S_l \in t_j$
    $A \leftarrow A \cup \{S_l^{new(i, q_{S_k^i})}\}$
    $q_{S_l}^{new(i, q_{S_k^i})} \leftarrow q_{0S_l}$
    $R_{S_l}^{new(i, q_{S_k^i})} \leftarrow \{\}$
    $v_{S_l}^{new(i, q_{S_k^i})} \leftarrow v_{S_l}$
  else goto step 2

4. When a shuffle automaton S_k^i in the active state reaches one of its final states it returns to its calling automaton, is removed from the active set and updates its parent's state vector

if $\exists i : S_k^i \in A$ **and** $q_{S_k^i} \in F$
then
  $A \leftarrow A \setminus \{S_k^i\}$
  if k=0
  then
    if $F_C.v_{S_0^0}$ **then accept else reject**

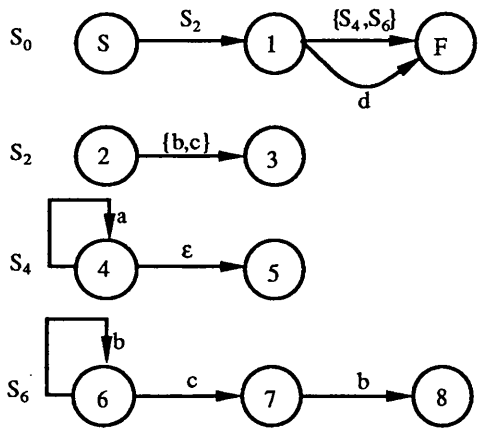

Figure 6.10: Automata generated for EPS 'Example'

$R_{S_k^{parent(i)}} \leftarrow R_{S_k^{parent(i)}} \cup S_k$
forall $i : 1 \ldots m$ **if not fixed** $v_{S_k^{parent(i)}}[i]$ **then** $v_{S_k^{parent(i)}}[i] = v_{S_k^i}[i]$

5. goto step 3.

This algorithm offers a compromise between behavioural immediacy (UCSAs fail immediately if an unwanted event occurs) and investigational specification (if a constraint fails, parsing continues to complete unification).

Consider, as an example the stylized EPS:

```
EPS: Example
''Example specification to show how UCSAs parse events.''

satisfies  tr
inwhich    b|c,(a*n||b*m,c,b)\/d
iff        n>4
```

The eight event alphabet and event parameters have been omitted for brevity. This specification generates a 'root' machine with three subordinates as depicted in Figure 6.10

The automata values are, for the 'root' machine:

$$\Sigma = \{a, b, c, d\}$$
$$S = \{S_0, S_2, S_4, S_6\}$$
$$T = \{t_1 = \{S_2\}, t_2 = \{S_4, S_6\}, t_3 = \{b, c\}, t_4 = \{a\}, t_5 = \{\epsilon\},$$

$$t_6 = \{b\}, t_7 = \{c\}, t_8 = \{b\}, t_9 = \{d\}\}$$

$$A = \{S_0\}$$

$$Z = \begin{pmatrix} m \\ n \end{pmatrix}$$

$$U = \{I, \mathbf{fix}, = x, +x, -x, /x, *x, \mathbf{concat}x, \ldots\}$$

$$C = \{< x, \leq x, > x, \geq x, = x, \subset x, \ldots\}$$

$$F_C = \begin{pmatrix} I \\ (> 4) \end{pmatrix}$$

$$W = \{w_1 = \begin{pmatrix} I \\ I \end{pmatrix}, w_2 = \begin{pmatrix} \mathbf{fix} \\ \mathbf{fix} \end{pmatrix}, w_3 = \begin{pmatrix} I \\ I \end{pmatrix}, w_4 = \begin{pmatrix} I \\ (+1) \end{pmatrix}$$
$$w_5 = \begin{pmatrix} I \\ I \end{pmatrix}, w_6 = \begin{pmatrix} (+1) \\ I \end{pmatrix}, w_7 = \begin{pmatrix} I \\ I \end{pmatrix}, w_8 = \begin{pmatrix} I \\ I \end{pmatrix}, w_9 = \begin{pmatrix} I \\ I \end{pmatrix}\}$$

and for the subordinates:

$$Q_{S_0} = \{S, 1, F\},\ q_{0S_0} = S,\ F_{S_0} = \{F\},$$
$$M_{S_0} = \{m(S, t_1) = (1, w_1),\ m(1, t_9) = (F, w_9),\ m(1, t_2) = (F, w_2)\}$$
$$Q_{S_2} = \{2, 3\},\ q_{0S_2} = 2,\ F_{S_2} = \{3\},\ M_{S_2} = \{m(2, t_3) = (3, w_3)\}$$
$$Q_{S_4} = \{4, 5\},\ q_{0S_4} = 4,\ F_{S_4} = \{5\},\ M_{S_4} = \{m(4, t_4) = (4, w_4),\ m(4, t_5) = (5, w_5)\}$$
$$Q_{S_6} = \{6, 7, 8\},\ q_{0S_6} = 6,\ F_{S_6} = \{8\},$$
$$M_{S_6} = \{m(6, t_6) = (6, w_6),\ m(6, t_7) = (7, w_7),\ m(7, t_8) = (8, w_8)\}$$

6.4.4 Implementing UCSAs

UCSAs are readily implemented as C++ classes with the appropriate registers, mapping functions, sets and progression algorithm. Such parsers are slow to build from textual descriptions of EPSs, but they are highly manipulable. An alternative UCSA implementation technique involves creating a new parse tree node for each EPS operator and building an EPS parse tree at compile time. We have adopted the latter approach. Although it entails a higher communications bandwidth, it is easier to distribute subnodes, more compatible with existing Solve architecture and each node is simpler. Also, in using this approach, we improve the ease with which the EPS prover can delegate constraint evaluation to existing Solve parse tree nodes.

6.5 Implementation Status

Circumstances have precluded a full implementation of a parallel, object-oriented exception handling mechanism based on EPS. The lack of a parallel Solve system has meant that some parts of

the system have necessarily been implemented in early sequential prototypes of Solve, while others have been built in isolation to demonstrate their feasibility.

The syntactic changes outlined in Chapter 5 have been fully implemented and tested on a sequential prototype of the Solve system. In addition, all new parse tree nodes (except EPSNode) have been implemented and tested as far as the (somewhat imperfect) prototype allows. Both *resume* and *termination* models have been implemented *in situ*, while the others have only been tested using a method invocation cycle simulator. The interpreter of this sequential prototype has been fully instrumented. This instrumentation is currently implemented as C++ 'in-line' member functions, invocations of which are substituted, at compile time, for the defining code by the C++ compiler. This eliminates the overhead of function call and affords a high degree of efficiency. Currently, instrumentation exacts a 12% performance penalty on interpretation. Its cost on generated code is yet to be established.

Simple automata (both CSAs and UCSAs) have been implemented in ML and, of late, some have been translated to C++. The latter process involves the considerable overhead of supplying, in C++, types native to ML, while affording the benefits of better performance and integration prospects. We have found that the improved performance of 'direct' UCSA implementations over parse tree structures (see above) warrants the extra compile-time complexity, lending on average a 120% performance benefit. Generally the performance of UCSAs (as implemented) is acceptable, but not above the need for optimization.

6.6 Limitations

Despite their enhanced applicability to this application, UCSAs exhibit many of the weaknesses of CSAs. For example, no recovery from EPS failures is possible because of the difficulty of interfacing any UCSA manipulation (that might be necessary to return the parser to a state from which an 'accept' is possible) with Solve handlers. This reflects our reluctance to over-burden any host language with EPS concepts and our belief that violation of the temporal protocols embodied by EPSs indicates a very serious problem from which recovery is undesirable.

Clearly EPSs are not suitable for real time systems. Firstly because of the difficulty, even with a Lamport clock, of obtaining a globally consistent and accurate view of 'real-time' and secondly, because of the overhead of instrumentation and UCSA parsing.

Finally, UCSAs are unable to share events (see [Bat89]) because no method of determining when such sharing is desirable exists. Without sharing, one encounters the *assignment problem*. If

two sub-automata of a parallel automaton both have event x in their alphabets and event x occurs, which automata should have x? One must consider that this event might be of the same event stream as one of the automata, of a different one, or part of irrelevant event noise that has passed through the relevant trace filter. The solution to the latter dilemma is to strengthen the filter, EPS is better in this regard than Bates' EDL (see Section 7.2.2). The former can be addressed by using a metric of event suitability for each sub-automata—we do not address this issue. Problems abound however, even if such a metric can be defined: what happens if the suitability metrics for two competing automata are identical? Can suitability be determined for each event without crippling performance?

Chapter 7

Related Work

7.1 Object Oriented Models

We are not alone in our efforts to formulate a model which describes object oriented systems behaviourally, but as far as we are aware, little work pertaining to the formalization of the *operational* semantics of a parallel object oriented system has been undertaken. Many new models have appeared during the course of this research, many too late to have an influence on it. Here we compare some of these to our own work.

7.1.1 Operational Models

Nierstratz and Papathomas [NP90] have established a framework for translating object oriented syntactic constructs into patterns of communicating agents in CCS and CSP, allowing entire concurrent, object languages to be behaviourally prototyped. This framework is used to test the interference that occurs between the language concurrency model and inheritance. The authors conclude that insufficient information appears in type signatures to avoid violation of encapsulation in concurrent languages—signatures should contain some indication of when services are available, how they may be interleaved, the mutability of a type and how instances may change rôles[1]. Although the authors provide a full framework and an incomplete behavioural subtyping relationship to use with this framework, they do not provide a model *per se* and their goals (formalizing language design and modeling constructs) are quite different from ours (operationally specifying behaviour). This limits the grounds of comparison between the two works. Within the last few months, Papathomas has extended this framework and produced a full CCS based model

[1] Through a very different rationale, Solve already supports the former two.

of a concurrent, object oriented system [Pap91] in a bid to analyze feature interference. It shares many features of our model, but is more abstract (the synchronization constraints if its scheduler may be altered), has improved inheritance modeling and supports hybrid and pure object oriented languages. Unlike our model however, it does not have dynamic channel allocation, consequently objects have to be maximally interconnected (ruling out easy simulation). Also the model does not yield any event alphabet. Alas, as with other works mentioned here, this work has appeared rather too late to have any effect on our own.

In [NH86], Nguyen and Hailpern propose a typeless object oriented model that supports multi-dimensional inheritance, causing method lookup to be influenced by receiver, selector, arguments and sender—a more flexible approach than our model which uses only the first two. Like our model, objects are defined with communicating sequential processes, but they support only synchronous message passing. The model supports parallelism to the object level of granularity and notes several weaknesses in its own object encapsulation and invocation schemes, in contrast our model uses CSP's own encapsulation mechanisms which appear to be stronger. Nguyen and Hailpern's system supports many features which are very uncommon in object oriented languages, for example the inheritance of individual methods, which tends to fragment objects somewhat. The authors approach is very abstract, they provide few details concerning the model's algebra and refrain from giving any information from which an event alphabet might be extracted.

7.1.2 Non-Operational Models

Significant success has been achieved in the formalization of program specification and denotational semantics. By criticizing the earlier attempt of Reddy [Red88] and Wolczko [Wol88], and presenting his own abstract semantics for object oriented languages, Yelland [Yel89] provides a framework in which observationally identical systems always have equal denotations. Cusack [Cus89] describes a set-theoretic model of inheritance which enables it to be grafted onto CSP using change of symbol and conformance operators, although the underlying motive is to provide object oriented extensions to existing specification tools rather than to behaviourally model object oriented systems. The grounds for comparison between these works and our own is limited. They concentrate on the structure and semantics of sequential object oriented languages (although America considers parallel languages in [ADKR86, Ame89b]); our attention is focused on the *operational* behaviour of a parallel object oriented system. Although dependent on the computational model of a system, this is largely independent of language. De Boer et al. [dB90] have created a high level model for the behaviour of evolving pointer structures that comprise the run-time state of parallel, object oriented programs. This is part of a broader attempt to use Hoare-style proof techniques to es-

tablish partial correctness of POOL programs. The goals of this model are very different to those of our own; it focuses on the language semantics and is POOL specific, whereas our own work is centered on the behaviour of systems.

7.2 Behavioural Specification and Recognition

7.2.1 Pathrules

There have been several attempts to specify, describe and recognize event-based behaviours in sequential and parallel systems. The *Pathrules* language was adapted from Ander's path expressions formalism [And79], by Bruegge [Bru85, BH83], as a tool for the expression and testing of high-level bug hypotheses and the definition of required debugger functionality. Originally designed for specifying the synchronization points of concurrent processes, Pathrules is a powerful way of defining the behaviour desired of a parallel procedural system in terms of procedure invocations and the occurrence of certain states. Like EPS, Pathrules triggers certain behaviours if they are satisfied, for example suspending execution of the target process or enabling triggering of another, previously dormant, pathrule. As a result of this flexibility they can establish debugging contexts which would be virtually impossible with conventional lexical breakpoints.

Unlike EPS, Pathrules is a complex production system language which might itself merit a debugger. For instead of providing a flexible tool to be used with other debugger commands, as EPS does, Pathrules is used to 'build' debugger commands and the apparatus needed to support this can discourage casual usage. To define new debugger commands one must make incremental changes to the debugger's input parser, a clumsy, low-level and error-prone task, the cognitive load of which harassed programmers debugging their code are unlikely to relish [MWPC83]. In user trials of Bruegge's *P & A* debugger, the first to use Pathrules, less than 5% of users altered the debuggers functionality and the majority confined themselves to usage of facilities available in conventional debuggers. Unlike EPS, the Pathrules specification formalism does not offer support for hierarchical definition of higher order events (see Section 4.4.8), consequently specifications must always be in terms of primitive events or those manually instrumented. Furthermore, no formal support for Pathrules' event alphabet is offered. Compared to EPS, Pathrules' power of specification is limited because it does not offer any direct support for fuzzy matching or unification. Unlike EPS, Pathrules is based on finite state automata theory and is thus subject to the weaknesses covered in Chapter 6 and [Bat87b] with regard to recognizing parallel behaviours. Most unfortunately, the Pathrules system has two distinct goals: specification of behaviour and adding functionality to a debugger; much of the design of this system demonstrates the crosstalk between these goals. In

contrast, EPS has only one goal set which satisfy two distinct problems. Finally, EPS is designed specifically for the description of object oriented systems—whereas Pathrules' targets are inherently procedural (see Section 7.3).

7.2.2 EBBA

Like EPS, Bates' EBBA system [Bat87b, Bat89, Bat87a] is specifically designed to compare event-based user specifications with actual program behaviour and act on the differences. It is designed to ease debugging in loosely-coupled, distributed environments. EBBA uses a constraint shuffle automaton (CSA, see Chapter 6) to compare streams of events representing program behaviour with user templates derived using an event definition language *EDL*. In a similar manner to EPS, EDL is able to define higher order events, but is incapable of analyzing or matching behaviour using the internal state of a thread as EPS is, i.e. it lacks data events. The primitive events with which EDL builds higher order events must be produced by the manual instrumentation of user programs—no default event alphabet is supplied. In designing EPS we have deliberately avoided this approach for the reasons outlined in Section 2.4.11. While it promotes some flexibility in that the user can choose her own event alphabet (and greater efficiency since events that might otherwise be higher order can be instrumented directly), it does mean that code amendment must occur before debugging can commence and there is no guarantee that the event alphabet will be what the user intended, or necessary and sufficient to report all behaviour.

Unlike EPS, the policy of EBBA is to watch and report program activity without intervention. Despite the difficulties of distributed debugging that gave rise to this view, we feel that such a policy is limiting: during debugging one often needs to intervene during program execution to gather data about a hypothesis (see Section 2.4). EBBA's CSA based, behavioural recognition cycle 'coarse-filters' the event stream before allowing each event therein, which satisfies the constraints, to alter the state of a non-deterministic automaton towards a failure or a match. EDL has no facility to finely filter events as EPS has. Furthermore, this approach inherently prevents fuzzy matching and unification, because a constraint failure causes behavioural matching to fail immediately, limiting the power of specification. This limitation is built into CSAs because the transitions are constraint driven, our alternative (UCSAs, see Chapter 6) overcomes this by making unification driven transitions and checking constraints after such unification. Like Pathrules, EDL offers no support for the behavioural idiosyncrasies of object oriented systems.

7.2.3 MuTEAM

The MuTEAM [BFM+83] debugger is an event based system for a concurrent language based on CSP. Like EPS, MuTEAM allows program behaviour to be compared to operational specifications which themselves are designed around a formal CSP model. MuTEAM's model has both denotational and axiomatic components, whereas EPS's is operational. In contrast to EPS, MuTEAM specifications have an alphabet limited to inter-process communication events (equivalent to the EPS event classes *send*, *terminate* and process oriented versions of *alter* and *create*), severely limiting its ability to debug intra-process state changes. This is a deliberate decision by the authors, they feel that traditional debuggers can cover this functionality—we do not concur. MuTEAM fosters no permanent association between specifications and programs as EPS does. Like EPS, the MuTEAM specification system can specify partial ordering of events and event partiality (using the **ANY** wildcard which is much like EPS's '?'). However its unification mechanisms are weaker than those of EPS: it allows only single event unification (with a somewhat clumsy syntax). A novel feature of the MuTEAM specification language is its ability to offset the probe effect introduced by behavioural checking, by delaying processes given an unfair advantage by the system. However, the manual nature of this **DELAY** construct can introduce insidious errors into programs if not used carefully. The constraints offered by MuTEAM specifications are also weaker than EPS in that they may only express conditions in terms of the process state and event counters. Furthermore, MuTEAM has no action clauses—any specification violations cause a program to halt and user intervention is required.

7.2.4 DEBL

DEBL [CW89] is a specification medium, based on temporal event logic (TEL), for the debugging of parallel programs. Indeed, DEBL is merely a front end, all DEBL specifications are converted into TEL. Consequently, DEBL is able to directly express temporal relationships between events that EPS is not, for example the relation *eventually*. However, this enhanced power of specification is achieved at a cost: DEBL is a retrospective technique, specifications can only be checked on traces of executions that have finished. This restriction makes DEBL less useful in *live debugging*[2] and totally inappropriate for exception handling. Unlike EPS, DEBL's alphabet consists purely of inter-process communication events (equivalent to EPS's *send* event class) and consequently it is unable to specify behaviour within processes. DEBL supports some degree of partiality with its wildcard **ANY** which performs a similar rôle to the EPS '?' construct (see Section 4.4.3), but unlike

[2] In live debugging, as opposed to replay (see Section 2.4.13), users interact with and manipulate the state of active processes.

EPS it offers no unification facility and its specification constraints can be expressed in terms of event parameters only. In EPS the temporal applicability of specifications can be restricted by making them *strict* (see Section 4.4.6) or by disabling them; DEBL specifications must apply to the entire life of a process—limiting their flexibility. Although DEBL specifications are not associated with programs as EPSs are, their `clue` field formally associates each one with a knowledge base—promoting reuse.

7.2.5 Executable System Specification for JSD

The *Life-Script* facility of Kozaczynski's Executable System Specification for JSD [KJ88] bears some resemblance to EPS. Although not targeted at object oriented systems as EPSs are (indeed JSD is a very poor technique for object oriented design [Som89, Pun90]) and lacking any support for partial specification, ESSJSD does support temporal specifications of the protocols in which program objects may indulge. However, unlike EPS it is a design aid—its specifications do not pass into the implementation and are not subject to run-time checking. ESSJSD's specifications are, by the author's admission rather verbose and not separated from the main body of specification, thus losing the benefits of modularity and reusability that EPSs have. They are translated from Prolog, implying some dependence on the logic paradigm and are highly JSD specific. We have endeavoured to keep EPS specific to the host paradigm only.

7.3 Object Oriented Behavioural Specification

All of the specification systems mentioned thus far fail to address the additional behavioural complexities unique to the object oriented paradigm. This is a consequence of their historical background or ambit, rather than a failure *per se*. The event alphabets (where given) of procedural specification systems are inadequate for object oriented systems because they fail to describe behaviour completely and unambiguously. Completeness is compromised by an incomplete alphabet, missing events like *create* and *lookup* without which a complete understanding of behaviour is impossible and clarity by missing event parameters (e.g. the host class in the *execute* event) which render existing event classes vague or meaningless. The additional events and event parameters are needed to describe the new behavioural features of object oriented environments and those features embellished from earlier procedural systems. For example, the new effects of polymorphism, inheritance and instantiation and the amended variable scope, variable lifetime and binding rules.

7.3.1 PROCOL

Van den Bos' C based, object based language PROCOL [vdBL89] offers a 'protocol' facility that, like the EPS exception handling system, may be used to order and constrain inter-object communications and to control object access. Protocols are access control guards which define the conditions under which an incoming message will result in the execution of a method, in an exception handling context EPS has a similar ability. Like EPS, each object class defines many protocols. They can be used to specify iterations, selections and sequences of method executions and state expressions (guards) which must occur before a certain method is eligible for execution. PROCOL uses nondeterministic finite state automata (NFA) to represent protocols, unlike the UCSAs used by EPS, these are subject to the limitations listed in Section 6.4.2. Each successful method execution advances the NFA parsers of all the protocols defined on the host object.

The PROCOL system has a number of important limitations that are not shared by the EPS exception handling mechanism. Firstly, protocols are only capable of specifying the preconditions of a method execution, i.e. of *guarding* it, no postcondition facility is available. Furthermore, the alphabet of the protocols includes only one event, method execution, limiting the detail of specifications. The model (and substrate language) fails to incorporate inheritance or delegation (a crucial facet of all object-oriented languages [BGM89, Weg90, LMT89]). The PROCOL model also omits assignment (*access*) events (the value of which was established by [Bru85]). The only partiality permitted in PROCOL specifications is the inequalities of state conditions—specifications are not subject to unification, severely restricting their power. In addition, each protocol guards only one method, whereas EPSs specify a temporal relationship between all events mentioned in the specification. PROCOL is not capable of specifying parallel event interleavings, as EPS can. Although this is unnecessary because the host language only supports parallelism to object granularity, the limitation is so engrained in the design and implementation of protocols that it would be difficult to overcome if the language were ever enhanced to support method parallelism. Unlike EPS, once an error is detected by a PROCOL specification, no direct action can be taken by the receiver other than blocking until the error is resolved. This, coupled with the provision of object parallelism, means that mutually recursive methods can easily deadlock the system.

7.3.2 Specification for Subtyping

America [Ame89a] has developed an operational specification formalism for sequential object oriented systems based on preconditions, postconditions and a data typing language. With it, he establishes the basis of a mechanism to support behavioural subtyping, as distinct from inheritance, free from the implementation of objects. Unlike EPS, America's formalism does not reason

about sequences of events, but uses an approach based on mathematical domains which dilutes the operational flavour of the medium. America claims that the former approach is not sufficiently abstract, while this is possibly true for subtyping, the hierarchical definition of events facilitated by EPS overcomes this with respect to our goals (see Section 3.5). America's scheme is not tractable to run-time checking and requires a theorem prover to be fully viable. Also, it does not seem to be very useful in a debugging context. However, these criticisms do not detract from the usefulness of the formalism in subtyping and a strong case is made that domain based specifications are more 'natural' and rigourous than purely operational ones.

7.3.3 Data Path Debugging

Data Path Debugging (DPD) [HK89] is a system which, like EPS, is implemented to support a problem-directed approach to debugging of object oriented, concurrent applications. It is derived from the pathrules system (see Section 7.2.1) and extended to support *standardized data events* (assignment and state change conditions), message events and validation of sequential and concurrent paths. Indeed, DPD stresses the importance of data events to the virtual exclusion of control events which we feel is overly restrictive. DPD is also able to track data dependencies between threads for those language environments that explicitly use dependencies, and introduces new operators to distinguish between the new types of concurrency this entails.

The Data Path Debugging system shares with EPS the ability to monitor concurrent object behaviours as a stream of events and to report deviations of actual traces from those specified. Unlike EPS, DPD is solely a debugging tool for locating errors once their presence has been established by testing, DPD specifications are ephemeral and are not associated with types as EPSs are. The event alphabet of DPD differs from that of EPS in that it is informally derived and puts great emphasis on state change events, attempting to map all aspects of behaviour on to changes of state—an approach which is rarely feasible [LL89]. While it is true that many bugs are incarnated in inappropriate data values as the authors claim, the root cause is often a flawed flow of control—we consider that *both* aspects of the dichotomy must be addressed. Unlike EPS, some of DPD's functionality supports the behavioural *patching* of software, we feel this dangerous practice should not be encouraged and is, in any event, frequently obviated by the advent of modern incremental compilers (see Section 2.4.10). DPD's specification medium does not offer unification, as EPSs do, and its support for partial specifications is limited. However the former limitation can be partially overcome, at the price of increased verbosity, by the use of action clauses with side effects. A unique feature of DPD is its use of standardized multiple history graphs to check the

accuracy of behaviour—allowing behaviour to be analyzed both immediately and off-line and for determinism to be enforced by replaying histories.

7.3.4 Specifying Object Interactions

Several formalisms have recently emerged for the in-source specification of object behaviour in terms of interactions. Helm et al's 'Contracts' [HHG90] enable users to: specify object compositions; denote their type and causal obligations; and capture the patterns of behaviour and behavioural dependencies of object frameworks. This abstraction supports 'interaction oriented design' and allows framework invariants and instantiation preconditions to be specified in a semi-formal way (similar to Solve's domains and the first event of an EPS). Like EPS, it improves program and framework understanding by making concrete behavioural dependencies which would otherwise have to be inferred from the source code. Like EPSs, Contracts can be built from each other using refinement and inclusion. However, above all, Contracts constitute a framework composition mechanism, a method of factoring out the complexity of frameworks, rather than a specification medium and this distinguishes them from EPSs. Although the specification language does support a subset of EPS functionality (with the equivalent of *send* and *access* event classes and the sequence, state check and interleaving operators), none of the specifications are checked at compile time, nor are the Contract invariants monitored at run-time. The conformance declarations produced from contracts are semi-formal and, as the author admits, cannot always be verified. Also, unlike EPSs, Contracts can violate the encapsulation of co-participants within an instantiated framework.

In [Ara91], Arapis presents a method of specifying the temporal properties of an application using first order temporal logic (FTL). Arapis asserts, as we have, the importance of temporal properties and the need to reason about execution sequences. The event alphabet used by his system is equivalent to the EPS event classes *send*, *execute*, *terminate* and unparametrized versions of *create* and *destroy*. Oddly, events are not parameterized by class or type as they are in EPS, resulting in a unique predicate for *create* as it applies to each class. This makes the predicate alphabet vast. Furthermore, it is supplemented with a rich set of data predicates which are defined temporally—making some of them very verbose. For example, consider the *class(a)* predicate, which is true if there exists an instance of class *class* called *a*. It is defined as the past creation of *a* of class *class*, without subsequent destruction of *a*. Arapis model is based around three entities: objects, activities and environments—all are highly analogous to each other and it is unclear why they are not unified into one compositionally recursive structure. Although all the examples shown in the paper could be specified using EPS, FTL undoubtedly constitutes a more abstract

and powerful specification medium—albeit less readable and easy to use. FTL specifications can be checked for mutual consistency (a very CPU intensive activity), but Arapis presents no means of checking specifications at run-time. Indeed this may be impossible as temporal logic specifications need to be checked against complete execution traces [CW89, GKY89], suggesting that FTL is useful as a post-mortem technique but quite unsuitable for exception handling or impromptu debugging (much like DEBL, see Section 7.2.4).

7.3.5 Behavioural Inheritance

Reghizzi et al. [RdPG91] propose the use of *behavioural inheritance* (*b-inheritance*) to avoid object inconsistency and to ensure high performance at run-time. Using multiple inheritance, an unconstrained (free) concrete class is combined with a generic *b-class* which constrains its behaviour through the use of Deontic logic (with a semantics and syntax based on pathrules, see Section 7.2.1 above). This promotes a more flexible, if less direct, association between specifications and class definitions than EPS. The rigour of deontic specification is improved by basing it on extended Petri nets, just as that of EPS is enhanced by basing it on CSP. Like EPSs, B-classes constrain the concurrency paths available to an object and its resource usage characteristics, typically ensuring mutual exclusion, alternation or limited parallel invocations of methods. They may be parameterized, enhancing the genericity of the specification medium. Deontic logic is only able to constrain behaviour and uses an event alphabet equivalent to the EPS event classes *send*, *execute* and *terminate*. State checks may be performed, without violating encapsulation, by *guards* which are designed to use only the exported methods of each class. The EPS state condition offers a similar functionality, although b-inheritance guards are somewhat more limited as they may only be checked at the termination of a method.

The usefulness of b-inheritance is compromised by a number of serious limitations. Firstly, b-classes may only be combined with concrete classes, they may not constrain abstract classes or be composed with each other to strengthen a specification as EPSs can (in each DAG path of an inheritance hierarchy, only one b-class may appear). This restriction on the use of b-classes will undermine the orthogonality of any language into which they are introduced. The authors state that by separating specifications from the class they constrain, class proliferation is avoided as multiple versions of the same class with different constraints can be easily accommodated. Since the composition of a free class with a b-class produces another class this argument is clearly invalid, although a number of other advantages of the approach remain—particularly specification reuse, abstraction and separation. As Deontic logic is based on pathrules it is subject to many of the limitations noted above in Section 7.2.1. In the proposed implementation, b-classes are

preprocessed into Ada, it is unclear how this effects debugging of specifications or the ease of changing them. The authors do not explain what happens if a specification is violated, but the specification medium has no equivalent to handlers or action clauses.

7.4 Exception Handling Systems

A wealth of object oriented languages and systems have included (or are including) exception handling mechanisms within their design [DPW91]. A small subset of these is addressing the additional problems of supporting exception handling in a parallel environment. None of these systems allow the definition of exceptions as behavioural violations as Solve does, but all offer some unique feature that is related to, has influenced, or would have influenced the design of Solve (had they been released earlier).

Many of the languages covered here represent signals, exceptions or both as objects. The advantages and disadvantages of this approach are addressed in Section 5.4.1. Predominantly, they include provision of a encapsulated medium to transfer exception information from the signaller to the handler context and a means of establishing default handling behaviour for certain exceptions. It is interesting that in at least one parallel language, *Guide*, such a representation is considered inappropriate.

7.4.1 ObjectWorks Smalltalk

The exception handling system of ParcPlace's ObjectWorks Smalltalk [DPW91, SM88], is the first associated with Smalltalk to have a traditional stack oriented handler search discipline [Goo75]. Like Solve, handlers are methods, but they may be associated with any closure. Class based association of handlers is not supported. Handlers support the propagation, termination and resumption control flow models. Unlike Solve, it is possible to stipulate, when an exception is signalled, that resumption is unacceptable. The exception handling system is not composed from language primitives as Solve's is, but designed around two classes: *Signal*, instances of which represent anomalous conditions; and *Exception*, instances of which represent a particular exception. Exception/handler associations are made and exceptions signaled by sending messages to an instance of Signal, this offers more flexibility than the Solve linkmap (see Section 5.4.4)—albeit less disciplined. As in Solve, even hardware errors are mapped onto signals to enhance congruency. Each Signal may have a parent and the exception hierarchy thus formed can be used to associate a handler to more than one exception. Each signal contains a set of default handlers, including those to execute

if exception handling itself fails. In contrast to Solve, all handlers have a single argument: an instance of *Exception* encapsulating information about the context of the exception. This instance holds all the information pertinent to the cause and environment of the exception.

Despite considerable flexibility, this scheme has a number of deficiencies which have come to light after some experience of its use [DPW91]. Divorcing the inheritance hierarchy from the exception hierarchy causes considerable loss of expressive power—although it does prevent class proliferation that might otherwise result. We feel that linkmaps provide a more elegant solution to the problem of *n:1* signal/handler mappings. Deutsch [DPW91] concludes that the resumption control flow model is unsound, chiefly on the grounds of its inefficiency and lack of safety. He prefers instead to promote the retry model. We concur with this view, but resumption is useful in some circumstances (i.e. within an assertion based system such as EPS, when the handler can be sure of providing a result which satisfies all the originally violated assertions). In Smalltalk, because a handler is unaware whether the failed method will be restarted or not, one must use a separate mechanism to perform cleanup. This mechanism, called 'unwind protection', triggers a cleanup method (chiefly to deallocate resources claimed by the method) if it detects the winding back of the invocation stack of the failed method. In parallel environments this approach is limited by the requirement for atomicity between claiming of resources and ensuring their deallocation in the cleanup method. Solve overcomes this problem by allowing handlers to establish the control flow policy.

7.4.2 Eiffel

The Eiffel exception handling system [Mey88, Mey89, Ner91, DPW91] has many novel features which have inspired later systems, including our own. It is the first system (and indeed, to our knowledge, the only system apart from Solve) to integrate exceptions and assertions within a rigourous *contract metaphor* and to allow implicit signalling. The main appeal of Eiffel's approach is its adherence to a few simple rules. Exceptions are abnormal run-time events: the violation of an assertion, attempted access to a void reference[3], method invocation failure (a violation of contract between client and server) or a hardware error. Unlike Solve, Eiffel exceptions are anonymous. In response to an exception an Eiffel method, after modifying the host object state as dictated by a method's `rescue` clause (an optional construct serving as a dedicated handler for the method), terminates (raising an exception within its client) or attempts to retry execution. Such a disciplined scheme ensures that software performs its task correctly, according to its contract, or not at all.

[3] A condition in which a variable references nothing, such that referring to it is legal, but de-referencing it is not. Solve variables cannot enter this condition.

As with Solve, the integration of software exceptions, operating system failures and hardware malfunctions greatly enhances the elegance of the Eiffel language.

The simplicity of the Eiffel mechanism has several drawbacks. Because exceptions are anonymous, one rescue clause in each method must handle all possible failures of that method. This hampers readability and restricts the ability of handlers to provide context sensitive solutions. Unlike Solve, such handlers may only adopt a propagate or retry control model and in the event of the latter, any alternative strategy for satisfying the contract must be co-located with the method body giving poor separation. Within a handler, control flow policy is fixed at compile time—one may not use conditions, expressed in the host language, to determine the best control flow model to adopt (as one can in Solve)—this is a severe limitation. Furthermore, there is no mechanism to propagate any information about exceptions, consequently the rescue clauses of classes high in the compositional hierarchy must address so many possible exceptions that they are restricted to the most general of responses.

In an attempt, to circumnavigate these failings, Eiffel has the class *Exception* which may be used to refine rescue clauses. This facilitates a catch/throw exception mechanism [Goo75]. Exception type, name, originating class, originating method and cause are made available through instances of the Exception class and specific exceptions can be ignored, caught, raised and handled through the mechanism which provides some of the flexibility of Solve's linkmap construct. However, the facility requires (counterintuitively) that the target class inherit from class Exception to reap these benefits. Furthermore, unlike Solve, the availability of exception related information is unrestricted, constituting a grave violation of encapsulation that undermines Eiffel's inherent discipline. Finally, the availability of a 'two-tier' exception handling service does diminish the orthogonality of Eiffel.

7.4.3 BETA

The flexible and highly original exception handling system of the Mjølner BETA system [KMMPN87, MMP89a, MMP89b, KM91] marks a significant departure from the classical approach proposed by Goodenough (see Section 2.3.1). We share with the authors the desire for a more rigourous exception handling system, but we feel that the price they have paid for this rigour is too high. However, the system comprises many good ideas, some of which have been adopted by our system.

BETA exception handling is unusual in that the extent of exceptions are defined by static structure of the code, there is no search for a handler at run time—all handler invocation is bound at compile time. This can not be fully done in Solve because of the possibility of dynamic binding. This renders BETA programs more tractable to formal verification. Handlers may be associated

with classes, programs, specific instances and method invocations (the latter is achieved by passing the handler as an argument when the method is invoked). Solve disallows all but the first of these to promote readability, consistency and discipline—but the flexibility of BETA's approach is an advantage. Handlers are named, executable objects which, unlike those of Solve, are signalled by explicit invocation. Through the use of *sequels*, handlers on the invocation stack are traversed from the bottom to the top (without altering the stack) and *then* again from the top down (while discarding the contexts therein), enabling the handlers in the hierarchy to interact and refine each other in ways impossible in more conventional systems. Resume, retry, spawn debugger and termination control models are supported, as is '*non-local goto*' to a specified label. Unlike Solve, termination is the default case. In contrast to Solve, no special mechanisms or syntax are provided for exception handling, one uses BETA conventions directly to deal with run-time anomalies. Consequently, programs that do not handle exceptions experience no overhead. Separation of conventional and error handling code is very well supported and all handling policies are instigated by sending messages to instances of the class *Exception*.

The syntax of BETA is extremely idiosyncratic, using terms like 'pattern' in place of 'class' and replacing much of the existing terminology accepted by the bulk of object oriented programming (OOP) researchers (i.e. those coined by [Gol83, Ren82, LSAS77, Coo86, Cox86], established by a succession of OOPSLA and ECOOP conferences and consolidated in [Weg90, BGM89]), without any apparent reason. This, allied with the lack of keywords used to support exception handling, the proliferation of symbols as reserved words, the use of non-local goto to support propagation, the two way flow of control imposed by sequels and the fact that individual instances of an object can have their own handlers, makes BETA source very difficult to read and understand for the uninitiated—even if they are familiar with the concepts of object oriented programming. Solve attempts to avoid this by adherence to traditional OOP terminology and copious use of extra keywords to aid readability. In BETA, the run-time search for handlers (in response to an exception) used by traditional systems, including Solve, is obviated at the cost, in certain circumstances, of placing checks for the handler bindings of superclasses within the source code itself (using the ## notation). This is a messy and potentially dangerous technique. The static binding of handlers in BETA also causes some counterintuitive behaviour. For example, cases in which a method invocation without a handler argument results in exception are not referred to the caller, instead the server class's handlers are used.

7.4.4 Guide

In [Lac91], Lacourte describes the synthesis of a statically typed, persistent, concurrent, object oriented language for a distributed system, Guide, with a classical (*Goodenough-esque*[Goo75]) exception handling mechanism. Like Solve, the mechanism totally unifies hardware and software exceptions. This is achieved in Guide's case by forcing programs to interface with the operating system through the virtual object *SYSTEM*. Guide actively supports separation of conventional and error handling source code (as Solve does) and employs a sub-mechanism to ensure object consistency under abnormal conditions. As in Solve, exceptions are symbols, *not* objects, in Guide and the author asserts that this vastly simplifies program control flow. Handlers may support the termination, propagation, retry and exit models. Unlike Solve, Guide handlers do not support resumption. The author asserts that such a facility requires closures to avoid violating encapsulation (using more than exported interface of object) and Guide does not have this construct. Exception/handler mappings may be associated with a method, a class, individual instructions or a block thereof, Solve does not support the latter two cases as we consider that they compromise the object oriented model. Like Solve, Guide defines these mappings in great detail, considering the type, name, host method and class of a signal. Lacourte distinguishes between the ambit of a mapping (as signified by the block delimiters) and the return point for a non-terminating handler. The return point is always just after the errant method invocation, independently of the deployment of delimiters. Solve also makes this, otherwise unique, distinction. Unlike Solve, Guide provides a series of default handlers for a range of situations (including the receipt of an exception while handling another), most of which cause method termination or propagation of a generic exception (to avoid *blind propagation* see Section 4). In addition, Guide provides a unique mechanism to determine the success of parallel compositions (using its `co_begin` and `co_end` operators). Each thread is assigned a boolean label which returns true only if it terminates successfully. Each parallel composition is accompanied by a boolean expression in terms of these labels, expressing the success criteria of the entire composition—failure causes the signalling of a special exception. Much the same effect can be achieved using hierarchically defined EPSs in Solve.

Like Solve, Guide does not use objects to represent signals or exceptions and consequently some other abstraction is required to convey the context of the former to the handler. Guide uses 'out' variables to do this and because access to these (unlike Solve's *dispatcher primitives*) is unrestricted, they can violate encapsulation. Furthermore, Guide's processing of hardware exceptions is crippled by the synchronous nature of its signalling mechanisms. Indeed, hardware exceptions must be converted into synchronous signals, severely limiting the immediacy of the handler. Unlike Solve, Guide offers no support for the debug, delegate or resumption handling models and its retry handler

may *not* be iteratively invoked—a limitation considering that the retry model is frequently used in situations where a bounded series of repeated attempts is the best recovery strategy (e.g. network communications failure, disk errors and user interaction).

7.5 Summary

Formal models of object oriented systems show wide variation according to their purpose. The author knows of *no* general purpose one. Many of these models are language oriented, being either denotational or operational perspectives of language construct semantics. Only recently has an operational model of a general language emerged, other than our own.

There have been several attempts to specify, describe and recognize event-based behaviours in parallel systems, but only comparatively recently have these been used, as we have used them, as part of in-language specification constructs. None have been applied to both debugging and in-source exception handling, and many (particularly those based on temporal logics) are inappropriate for these tasks. Early, procedural formalisms typically demonstrate lack of complete or rigourously defined alphabets, lack of selectivity in the applicability of specifications (filtering) and weak, non-unifying constraints. Specification media for object oriented systems have inherited these faults and are either incapable of doing anything if violations are detected, or unable to detect them at all. An increasing number of researchers are realizing the benefits of being able to reason about the operational behaviour of objects in their signatures.

There is much variation between exception handling mechanisms, but none yet use behavioural specifications to detect exceptions. Of late, some attempts have been made to formalize exception handling to cope with the rigours of parallel programming. We hope to have contributed something toward this end.

Chapter 8

Conclusions

8.1 Summary

We have established the feasibility of using operational specifications to detect and locate bugs in parallel, object oriented systems. This has been achieved by: extensive survey, to reveal how specification is already used in the detection and correction of program bugs and in what ways these techniques are deficient; modeling the behaviour of a parallel, object oriented system through the use of a process calculus; using this model to devise a formalism to operationally specify the behaviour of such a system; and successfully designing and implementing an exception handling system based on this formalism.

Having described three common ways of reducing the presence of bugs in computer programs and conducted an extensive survey on the types of mechanisms used to support two of these methods, exception handling and debugging, we indicated several serious weaknesses with these mechanisms. Firstly and most significantly, we found that techniques commonly used to support exception handling lacked the power of specification required for detecting errors particular to object oriented or concurrent systems. Secondly, many debugging tools for object oriented systems failed to fully support the abstractions of that paradigm. Thirdly, the few debugging tools supporting specification facilitated only the most ephemeral of associations between these specifications and their targets, giving users little incentive to use the technique to its best advantage. Most of the tools and mechanisms surveyed lacked formal rigour. We sought to overcome these problems by devising a formalism to describe the behaviour of parallel, object oriented systems and using it in an environment which facilitates permanent association of specification and source code.

We established a CSP based model of a general purpose, object oriented system with inheritance and asynchronous communication. We then determined, using this model, that the behaviour of

such a system can be completely expressed using an alphabet of eight events. This alphabet, while constituting the events one might have expected from informal analysis, could be used with confidence for formal behavioural analysis, without fear of loss of coverage, ambiguity or redundancy.

A formalism, called EPS, was derived from this model, for the hierarchical description of intra-object and inter-object behaviour in parallel, object oriented systems. This comprehensive, operational specification language supports partial specification, specialization, readability, reuse and action clauses that may be triggered depending on the fulfillment of these specifications. The formalism is language independent and supports exploratory specification by deferring the evaluation of constraints until pattern matching has terminated, so that the results of the latter and reasons for any mismatch can be ascertained. Details of how this formalism might be used as the basis of a debugging tool, and in more detail, how it might facilitate behavioural exception handling were discussed.

We have designed and implemented a disciplined exception handling system for a parallel object oriented language. The mechanism, which uses the EPS formalism, has a significantly greater power of expression and rigour than conventional counterparts without sacrificing readability. Indeed, by annotating source code with behavioural specifications, readability and ease of understanding are both enhanced (as explained in [Gol87, SBK81]). The mechanism emphasizes object oriented abstractions and permits a more holistic, inter-object specification—not possible with state based assertions. It does not impose a fixed exception handling policy; instead users may select one of six disciplined policies depending on the conditions prevalent at handling time. It integrates hardware and software exceptions and supports implicit, asynchronous exceptions which enhance the immediacy of handlers.

We demonstrated the feasibility of the exception handling system and thus of EPS, by implementing it atop the Solve programming language. The event parsers were constructed from a new type of constrained shuffle automaton—the UCSA—that enables sets of events from concurrent event sources to govern state changes and which defers constraint checking until after a behavioural match to facilitate unification and explorative specification. Unlike finite state machine based parsers, our implementation can accept symbol *sets* to control states changes and may be used to specify *any* concurrent behaviour.

8.2 Contributions

We have shown that operational specification is a feasible technique for the expression of behavioural assertions in parallel, object oriented systems—that is the main contribution of this

work. It may be used in a debugging context to make behavioural hypotheses or obtain behavioural information and, perhaps more importantly, it is a viable means of making behavioural assertions as part of an exception handling mechanism, that may be checked during execution. It offers a more rigourous and flexible means of specification than any other debugging tool or exception handling mechanism known to the author; thus providing a more immediate and direct means of detecting behavioural deviations.

This work makes four other contributions to the field of object oriented programming and language design. It describes, through a survey, the principle weaknesses of current debugging tools and exception handling mechanisms; it provides a formal model for the analysis of parallel object oriented systems; it presents a formalism for describing the behaviour of such systems; and details the design and implementation of a behavioural exception handling mechanism, including a new type of automaton.

8.2.1 Problems Revealed by the Survey

We have, during our survey of such systems, uncovered several problems and deficiencies prevalent in debugging tools and exception handling mechanisms. Some of these problems are rendered remarkably obvious when a model of a general purpose debugger (or exception handling mechanism) is used to structure comparison. Aside from the problems addressed directly by this thesis (see Section 8.1), debugging tools were found to: offer poor source visualization, provide inadequate support for impromptu debugging, have limited techniques for associating run-time behaviour with source text and show excessive dependence on the host architecture. Exception handling techniques were found to be: ill disciplined, poorly integrated with analogous schemes for hardware exceptions and inadequate at separating main and handler source code. These are all issues which need to be addressed by language and tool designers.

8.2.2 Formal Model

We have demonstrated how to model an object network in CSP without using maximal interconnection: by facilitating dynamic channel allocation. This, in theory, enables our model to be tested on the various CSP simulators that exist (e.g. [DS86]), although we have not done so. Furthermore, we have demonstrated how one can model asynchronous communications, within CSP (which is inherently synchronous), through use of a message bus and object mailboxes. The model has demonstrated the need for method typing (particularly for the message type *error*) in systems not supporting RPC.

Although this model has some limitations (see Section 8.3), it serves as an example of the way in which object oriented systems can be operationally modeled and illustrates the advantages of such a process. It also demonstrates how two formalisms like CSP and Z may be used in unison to satisfy both the process based and data based requirements of systems modeling.

We have shown that the operational behaviour of a parallel, object oriented system, isomorphic to the model presented in Chapter 3, can be completely represented as a partially ordered sequence of eight parameterized event types. This *alphabet* is essential to the completeness of behavioural descriptions and specifications of object oriented systems. Furthermore, it may be used as the basis of instrumentation design to monitor behaviour, as the core of a program visualization facility, as the beginnings of a formalism to specify required behaviour (as it was in this case), or in the design of any medium associated with the behaviour of an object oriented system.

8.2.3 Behavioural Specification

We have designed a formalism called EPS, which is based on CSP, for the hierarchical description of behaviour in parallel, object oriented languages. This language is unique in that it may describe *any* behaviour of which a parallel, object oriented system (as defined above) is capable. Furthermore, it can describe it at various levels of abstraction and partiality. This is achieved by combining unifying, hierarchical pattern matching (which uses the event alphabet and CSP operators) with boolean constraints. This formalism is independent of the host language and requires only that it adhere to the model described above. With this language it is possible to achieve debugging contexts which cannot be achieved using traditional techniques, such as breakpoints. Moreover, it may be used to detect exceptions that state based assertions would miss or detect too late. The applicability of a single language in both of these areas is, as far as we know, unique to EPS.

We have established the benefits of separating the pattern matching elements of behavioural checking from the data constraints, to allow explorative specification—so called 'fuzzy' matching. This allows EPS to be used as a selective information gathering tool, in addition to its other uses.

One of the main goals of EPS—ease of use—has been supported by avoiding traditional formal methods. An example of visual specifications is given to illustrate how such a formalism may be made easier to use by programmers who lack the knowledge or inclination to use formal specification techniques. We believe that the unique qualities of graphical media (as discussed in Section 2.4.12) make an ideal specification medium.

8.2.4 Exception Handling Mechanism

We have demonstrated that it is feasible to build an exception handling facility based on behavioural specification. Furthermore, it is possible to graft this feature on to an existing parallel object oriented language without amendment of the existing features of that language.

This design demonstrates the benefits of asynchronous exception detection, i.e. unifying software and hardware exceptions and increasing the immediacy of handler response. The mechanism offers a set of seven disciplined handling strategies, ensures that all exceptions are handled in a disciplined manner and that any attempted recovery is complete with respect to the original remit of the errant object.

As part of the implementation, we have developed a Unifying Constrained Shuffle Automaton (UCSA) to facilitate the run-time parsing of collected events. These events are gathered by instrumentation of the interpreter to generate events within the formally derived alphabet. This is guaranteed to provide a complete view of system behaviour, including control and data events, without need to manually instrument code. UCSAs facilitate the gathering of events from concurrent sources, but do not prematurely reject pattern matching due to the failure of a local constraint (as Constrained Shuffle Automata do), continuing instead to collect more information about the details of the deviation. They offer greater efficiency than CSAs and better support for 'fuzzy' pattern matching. During tests of our implementation on the Solve interpreter, we have found the overhead of instrumentation and parsing to be significant, but not unacceptable. However, this overhead would preclude the use of EPS in real-time systems.

8.3 Limitations

8.3.1 The Object Model

Our CSP model has some deficiencies which limit its efficacy under some conditions. It was designed to model *pure* object oriented systems (like Smalltalk or Solve) in which all program entities are objects. However, many object oriented languages are hybrid (e.g. C++, Objective-C) and use *builtin* types (typically integers, booleans, reals...) to optimize performance. These types are not manipulated like objects and this lack of orthogonality, overlooked by our model, may effect the event alphabet and behaviour of such systems. Further, our model does not consider the effects of automatic garbage collection, metaclasses, state access serialization (e.g. monitors), classes as objects or exception handling. These and other limitations are discussed in Section 3.6.

8.3.2 The EPS Formalism

We have not established that EPSs can be tested for internal integrity, consistency and meaningfulness, beyond ensuring that valid UCSAs can be constructed from them. Furthermore, there is no proof that EPSs can be checked for conformance against a stronger specification, as might be required in a object oriented language when an inherited specification is overridden. Both are beyond the remit of this work. As the formalism is based on CSP these properties could, in all likelihood, be inherited from the calculus—but we can not assert this without further work.

Some specifications can have several representations in the EPS formalism. In general, it is not possible to test for equivalence—a severe limitation if analogous objects are being behaviourally compared. Equivalent specifications frequently have different characteristics (see Section 4.4.7), they may have different immediacies and instantiate different numbers of variables. In the former case, if the user chooses the wrong representation and the specification triggers an action clause, it may execute too late to have any useful effect. There is no way, to our knowledge, of automatically rewriting specifications to ensure some property, or of warning users that a specification has low immediacy.

EPSs may only append events to the event queues of other EPSs—the queues cannot be edited or partially deleted. Such a feature would enhance the flexibility of the system, perhaps at the cost of additional complexity. A queue manipulation facility would also allow recovery from behavioural exceptions. Currently, one cannot resume, retry or locally exit execution in response to a failed EPS. This constitutes a lack of orthogonality (since one can with all other exception types) and flexibility.

To ensure that the specification medium is language independent, EPSs have no knowledge of the semantics of state assertions, as if defers evaluations of these assertions to the host. Consequently, they are unable to check that these statements do not have side effects on the host system. If the latter has 'pseudo functional semantics' (that is all message sending has no side effects), as Solve has, the problem can be reduced to one of ensuring the user does not alter any bindings inside such statements.

8.3.3 Exception Handling in Solve

The flexibility of type definition in Solve could be greatly enhanced if a class's behavioural commitments were separated from the protocol and type checking information contained in the signature. This would facilitate the factoring of behavioural information from the interface of a class, enabling

greater genericity and overcoming the overloading of the Solve class signature construct discussed in Section 5.6. Ideally, behavioural (type) information and signature (class) information should have separate inheritance hierarchies to reflect the different ways in which they are specialized.

As they are currently implemented, EPSs have a significant overhead. This is acceptable in a debugging tool, where such overhead is the temporary price of the enhanced informational and manipulatory control rendered by the debugging agent, but exception handling is a permanent feature. Consequently, some means of limiting this overhead must be found if they are to achieve wide-spread usage.

Event pattern specification is not suitable for use in real-time systems. Despite the availability of logical clocks [Lam78], the run-time overhead of UCSAs is such that it is not feasible to use them in such systems. Indeed, the overhead of instrumentation alone precludes their use due to the probe effect.

The Solve exception handling system cannot propagate a signal if an asynchronous method invocation encounters an exception after its parent dies. In short, there is no means of knowing to whom to send it. This is the *orphaned signal* problem [TL91, DPW91]. Several solutions exist, and inevitably, these raise further problems. One solution is for each thread to carry its own genealogy, allowing an exception delivery agent to try several generations of parents. However, to do this one must know if propagation is legal to these early generations. Another technique is to set up a central object that maintains a list of 'forwarding addresses' for parents of asynchronous children to leave notifying thread ids in when they die. However, this is time consuming to administer in a highly distributed system and relies on continued integrity of the central object itself—what would happen if this object received a signal!

8.4 Future Work

It is the author's fervent hope that this work has presented many possible paths of future research and experimentation. We have established the feasibility of this technique, but it has yet to be proven *useful* or implemented to its full potential. Much remains to be done.

Firstly, an implementation of these ideas on a fully parallel (and possibly distributed) substrate is required, preferably with dedicated hardware for EPS parsing. This will require careful attention to the orphaned signal problem (defined in Section 8.3.3) and others described above. Once achieved, the usefulness of the EPS formalism within an exception handling context could be determined by experiment and 'field trails'. Furthermore, the development of a debugger supporting

EPS (by combining UCSA's with a subsystem such as EBBA [Bat89]) would facilitate experiments to evaluate how the dual rôle of EPS mutually reinforce each other.

Currently, EPS constraints can only be expressed in terms of the behavioural or state based aspects of a program. By enhancing the domain of constraints to include static relations such as instantiation, inheritance and subtyping, considerable flexibility and power of specification could be achieved. Typically such predicates might check the genealogy of a class (i.e. *is_subtype($x,$y)*) or return the class name of an instance (i.e. *has_class($x)*). Such a concept would have to be implemented carefully to avoid introducing dependencies on the host language.

The user interface of EPS could be greatly improved by bringing some expertise in the psychology of programming and human-computer interaction to bear on the somewhat rudimentary visualizations presented in Section 4.5. By allowing both animated, event based displays of behaviour (for example, [BH90b, KG88]) and the formulation of specifications by *constrained* direct manipulation (of analogous visual representations of events), one might improve the ease of use of EPS (by circumnavigating the dull, error prone textual medium) and prevent the creation of internally inconsistent specifications. In previous software animation systems the event alphabet has not been considered formally, with EPS the completeness of the alphabet ensures the continued consistency between system state and a full graphical representation. It is tempting to hypothesize that graphical representations of specifications might enhance their abstractness.

One promising application of EPS is as a guide for semantic browsing algorithms. To promote reuse, one requires the compositional tools provided by object oriented programming environments like CoSIDE [Rob90], but one must also be able to locate reuse candidates in an object repository or library. By providing a browser that can conduct a search based on a behavioural template of what is required (textual or pictorial), as opposed to a class name or signature fragment, one is removing the current syntactic limitation on reuse and instead basing candidacy on behavioural conformance.

Event based behavioural definitions can be used to circumnavigate protocol incompatibilities between objects, facilitating reuse where otherwise it would have been impossible [VJN+90]. Event sequences of client objects might be mapped onto those of a server using the EPS formalism, and vice versa—thus overcoming protocol mismatches without altering the objects themselves. Here behavioural descriptions are used as the temporal 'glue' between the objects. The completeness of the event alphabet is essential to the power of this glue.

Object oriented programming obtains much of its power through abstraction, by raising the *level* of modularity of program sub-components. An obvious progression, for which there is some

evidence as noted above (see Chapter 7), is direct language support for the *metaobject* or framework. EPS already partially supports frameworks through the *send* event which may be used to relate or mutually constrain two objects in a framework. Clearly, as new languages support more advanced framework composition constructs, corresponding advances will be required of the accompanying specification medium. New constructs are required to specify how frameworks are instantiated, what degree of concurrency they support and their resource allocation behaviour.

Appendix A

Glossary

Anomaly. An unexpected error due to unforeseen behaviour of an agent beyond the control of the program, on which the program relies—for example hardware, or the user.

Accessor Methods. Methods defined on an object that, when executed, yield some aspect of that object's state *without* changing it [Mey88].

Assertion. A rule or truth equation which expresses an expected fact, the contradiction of which denotes an exceptional circumstance.

Bug. A defect in a program's specification, design or implementation that causes the latter to deviate from the expected or desired behaviour when executed.

Creator Methods. Methods defined on a type object that, when executed, produce new instances of that type [Mey88].

Chunking. Chunking is the cognitive process of associating a series of objects into one object, at a higher level of abstraction, to save short term memory [Mod79]. For example, one might abstract the concurrent letters 'c', 'a' and 't' into the single word 'cat'.

Exception. A class of states requiring extraordinary computation.

Handler. A code segment designed to gracefully terminate program execution or to mitigate the consequences of the bug or anomaly which triggered its execution.

Iconic Ligatures. A visualization technique, analogous to the hierarchical combination of primitive events into higher order events, in which icons representing primitive events are automatically combined according to a fixed set of rules based on the events involved and their temporal relationships.

Locality. In data structure or program design, the isolation of implementation details to avoid implicit dependencies between the defining and client objects. High locality localizes implementation changes and reduces the possibility that such changes will have undesirable side effects elsewhere in a system. First defined in [Lis87].

Particulars. In pattern matching, those components of behaviour and state satisfying the partial elements of a specification in a successful match.

Perpetual Match. In pattern matching, a condition in which a parser is able to continually satisfy its template. This occurs either because the template is cyclic, e.g. '`satisfies tr inwhich .?`' or not right bound (see above) as in '`satisfies tr inwhich .*`'.

Pre-Dormant. The initial state of a template parser in pattern matching. That state to which all EPS parsers are set when first instantiated.

Primitive Events. Instances of the eight event classes of Chapter 3, which represent the lowest level events available to EPS for describing system behaviour.

Post-Dormant. The state, entered by an EPS template parser, after it has resolved he success or failure of its specification.

Pre-Strict. In pattern matching, a template parser is pre-strict if it cannot refuse relevant events in its pre-dormant phase.

Post-Strict. In pattern matching, a template parser is pre-strict if it cannot accept any further relevant events in its post-dormant phase.

Right Bound. In pattern matching, a wildcard is said to be right bound if it has a defined end point, i.e. there exists a condition whereby the wildcard will be satisfied and its parser will attempt to satisfy the particle that follows it.

Signal. A value denoting an anomalous situation. Signals may represent specific types of failure or merely the concept of failure itself.

Signaller. A construct consisting of a signal (q.v.), an assertion (q.v.) and an evaluation time. When the signaller's evaluation time occurs, it checks if its assertion holds. If the assertion is found to be violated, the signal is raised indicating the occurrence of an exceptional condition.

Transformer Method. Methods defined on an object that, when executed, alter (transform) the state of that object [Mey88].

Appendix B

Usage of Mathematical Symbols

The following is a list of the mathematical symbols, complete with semantics, used in the definition of the CSP behavioural model (see Chapter 3).

Symbol	Source	Semantics
Σ (Sigma)	original	object space
σ (Sigma)	original	instantiation relation
ξ (Xi)	original	object binding name
η (Eta)	original	object instantiation function
ω (Omega)	original	object deletion function
$\mathcal{M}$	original	upper limit on number of simultaneous objects
$\mathcal{C}$	original	upper limit on number of simultaneously allocated BUS channels
$\mathcal{F}$	original	upper limit on number of simultaneously allocated future channels
$\cup$	trad	set union
$\in$, $\notin$	trad	set membership and inverse
$\subset$, $\not\subset$	trad	proper subset and inverse
$\setminus$	trad	set subtraction
$\equiv$	trad	is equivalent to
$\forall$, $\exists$	trad	universal and existential quantifier
$\Rightarrow$	trad	logical implication
$\Leftrightarrow$	trad	logical equivalence

Symbol	Source	Semantics
$f^\sim$	Z	functional inversion
$\rightarrow$	Z	total function
$\nrightarrow$	Z	partial function
$\twoheadrightarrow$	Z	total surjection
$\nrightarrow\!\!\!\!\twoheadrightarrow$	Z	partial surjection
$\leftrightarrow$	Z	binary relation
$\mapsto$	Z	maplet
$\oplus$	Z	functional override
$\times$	Z	cartesian product
$\mathbb{P}$	Z	powerset
Ξ	Z	the no change invariant
dom	Z	function domain
ran	Z	function range

Symbol	Source	Semantics
P/x	CSP	process P after event x
$\wedge$, $\vee$	CSP	specification conjunction, disjunction
α	CSP	process alphabet
$\gg$	CSP	process chain or pipe
$\mid$	CSP	simple choice operator
$[]$	CSP	deterministic choice
$\hat{}$	CSP	trace catenation
$\mathfrak{D}$	CSP	expression domain
$\{\}$	CSP	empty set
$\langle x \rangle$	CSP	the trace of the event x
$\langle\rangle$	CSP	the empty trace
x_0	CSP	the head of trace x
$\parallel$	CSP	parallel operator
$\mid\mid\mid$	CSP	interleave operator
//	CSP	subordination operator
!	CSP	channel output
?	CSP	channel input
$\hat{}$	CSP	process interrupt operator
#	CSP	trace length operator
$\lightning$	CSP	catastrophe event
$\emptyset$	CSP	null value
$\downarrow$	CSP	trace occurrence operator
$\overline{x}$	CSP	reverse of trace x
$\checkmark$	CSP	successful termination event
$\rightarrow$	CSP	then operator
$P \triangleleft x \triangleright Q$	CSP	if x execute process P else Q
x^n	CSP	event iteration
$\bigcup$	CSP	union of family of sets
sat	CSP	process satisfies specification
μ	CSP	bound variable of recursion
;	CSP	sequence operator

Appendix C

Solve Example

Here we present an example Solve type definition for a type parameterized stack, the formal syntax of Solve is discussed in Section 5.5. Although this example is somewhat contrived, it helps to illustrate many of the features of the Solve exception handling system.

In the signature `self` and `$x` are directives that refer to the instance receiving the message and the xth argument of that message send as they were *prior* to method execution. When decorated with a prime `'` such directives refer to the state of that entity *after* method execution. Primed directives may only be used in postcondition expressions (see Section 5.5). The term `result` represents the result of a message send, it is shorthand for `result'` (there is no result until the method has finished executing).

The signature declares one domain signal and defines one EPS signal, two precondition signals and four postcondition signals. The preconditions guard against invalid use of an empty stack and the postconditions check for integrity (*goodResult*) and that the stack size is changing as it should (*isLarger*, *isSmaller*). The EPS ensures that method invocation is serialized.

```
Signature Stack(element)
Supertypes (Object)

InstanceOperations

  push: (<element>) -> <Stack(element)>
    signal postcondition goodResult [ self' <-- top() <-- eq($1) ]
    signal postcondition isLarger   [ self  <-- size() <-- lt(self' <-- size()) ]

  pop: () -> <element>
    signal precondition notEmpty [ self <-- isEmpty() <-- not() ]
    signal postcondition goodResult [ self <-- top() <-- eq(result) ]
    signal postcondition isSmaller  [ self <-- size() <-- gt(self' <-- size()) ]
```

```
  top: () -> <element>
    signal precondition notEmpty [ self <-- isEmpty() <-- not() ]

  isEmpty: () -> Boolean

  size: () -> Integer

TypeOperations
  new: (<Integer>) -> <Stack(element)>

DomainSection
  signal sizeFault

TemporalProtocolSection
  correctUse
  "ensure stack is initialized and used correctly; disallow method parallelism"
  satisfies  tr restrict {execute, terminate}
  inwhich $i:execute($meth),$i:terminate($meth)

End
```

The matching implementation defines the size methods declared in the signature, the internal representation of stacks, the linkmap, the shadow methods and the domains of the object. The linkmap ensures that: all postconditions are handled by termination (by the shadow method *fatalproblem*); that if either the *notEmpty* precondition or the *feasibleSize* domain are violated, that a signal *notEmpty* is propagated (either of type domain or precondition); that any memory exhaustions that occur due to the use of the method *new* should result in up to four retries (each after a 400mS 'back-off' delay) before spawning a debugger to investigate the problem; and that any other signals (here, the violation of the temporal protocol) immediately spawn a debugger.

```
Implementation Stack(element)
Includes (Object)

InstanceSection

local storage <Array(element)>
local first <Integer> := 0
local MaxSize <Integer> := 100

export const push <Method((element), Self)> :=
  [ (item <element>) |
    storage <-- atPut(first, item);
    first := first <-- add(1)
  ]

export const pop <Method((), element)> :=
  [
    first := first <-- subtract(1);
    => storage <-- at(first)
  ]

export const top <Method((), element)>
  [
    => storage <-- at(first <-- subtract(1))
  ]
```

```
export const isEmpty <Method((), Boolean)> :=
  [
    => first <-- ge(0)
  ]

export const size (Method((), Integer)> :=
  [
    => first
  ]

local const initialize <Method((Integer), Self)> :=
  [ (size <Integer> |
    storage := Array <-- new(size);
    MaxSize := size
  ]

TypeSection

export const new <Method((Integer), Stack(element))> :=
  [ (size <Integer>) |
    let temp <Stack(element)> := self <-- super~new();
    temp <-- initialize(size);
    => temp
  ]

LinkSection

handles postcondition::*  with fatalproblem
handles precondition::notEmpty, domain::*  with underflow
handles postcondition::memoryLeft@Object::new with memoryfault
handles * with debugnow

HandlerSection

local const underflow <Method((), Void)> :=
  [
    first := 0;
    self <-- dispatcher~toclient(notEmpty)
  ]

local const fatalproblem <Method((), Void)> :=
  [
    self <-- dispatcher~terminate()
  ]

local const memoryfault <Method((), Void)> :=
  [
    System <-- wait(400);
    self <-- dispatcher~retry(4);
    self <-- dispatcher~debug()
  ]

local const debugnow <Method((),Void)> :=
  [
    self <-- dispatcher~debug()
  ]

DomainSection
  feasibleSize [ first <-- ge(0) ]

End
```

Bibliography

[Ada80] A. Adam. Laura, a system to debug student programs. *Artificial Intelligence*, 15:75–122, 1980.

[ADKR86] P. America, J. W. DeBakker, J. N. Kok, and J. J. M. M. Rutten. A denotational semantics for a parallel object-oriented language. Technical report, Center for Mathematics and Computer Science, Amsterdam, August 1986.

[AG89] Z. Aral and I. Gertner. High-level debugging in Parasight. *SIGPLAN Notices, SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):151–161, January 1989.

[Agh90] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[AH87] G. Agha and C. Hewitt. ACTORS: A conceptual foundation for concurrent, object-oriented programming. In *Research Directions in Object-Oriented Programming (B. Schiver and P. Wegner eds.)*, pages 49–74. MIT Press, January 1987.

[AM86] E. Adams and S. S. Muchnick. Dbxtool, a window based symbolic debugger for Sunworkstations. *Software Practice and Experience*, 16(7):653–669, July 1986.

[Ame87] P. America. A sketch for POOL2. Technical Report 0240, Philips Research Laboratories, Bedrijven, September 1987.

[Ame89a] P. America. A behavioural approach to subtyping in object oriented programming languages. Technical Report 443, Philips Research Laboratories, Bedrijven, April 1989.

[Ame89b] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(1):366–411, October 1989.

[And79] S. Andler. *Predicate Path Expressions: A High-Level Synchronization Mechanism.* PhD thesis, Dept. Comp. Sc., Carnegie-Mellon University, Pittsburgh, Penn., 1979.

[Ara91] C. Arapis. Specifying object interactions. In *Object Composition (D. Tsichritzis ed.)*, pages 303–322. Universite De Geneve., Geneva, Switzerland, July 1991.

[Bac86] L. Backhouse. *Software Construction and Verification.* Prentice Hall International, September 1986.

[Bal69] R. M. Balzer. EXDAMS – extendable debugging and monitoring system. In *Proceedings AFIPS Spring Joint Computer Conference*, pages 567–580. AFIPS, 1969.

[Bal84] L. Baldwin. Color considerations. *BYTE*, 9(9):227–246, September 1984.

[Bat83] P. Bates. Requirements/design debugging. *SIGPLAN Notices, Symposium on High Level Debugging*, 18(8):32–33, March 1983.

[Bat87a] P. Bates. EBBA modelling tool a.k.a. Event Definition Language. Technical Report 87-35, University of Massachusetts at Amherst, Mass., April 1987.

[Bat87b] P. Bates. Shuffle Automata: A formal model for behaviour recognition in distrubuted systems. Technical Report 87-27, University of Massachusetts at Amherst, Mass., January 1987.

[Bat89] P. Bates. Debugging heterogeneous distributed systems using event based models of behaviour. *SIGPLAN Notices, SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):11–22, January 1989.

[BB88] M. Baldassari and G. Bruno. An environment for object-oriented conceptual programming based on PROT nets. *Lecture Notes in Computer Science*, 340:1–84, January 1988.

[BEH88] T. Bemmerl, N. Erl, and O. Hansen. Menu and graphic driven human interfaces for high level debuggers. *Microprogramming and Microsystems*, 24:153–159, January 1988.

[Bei84] B. Beizer. *Software System Testing and Quality Assurance.* Van Nostrand Reinhold, New York, 1984.

[Bel89] F. J. Bell. A tutorial on the formal specifications of OCCAM programs. Technical report, University of Ulster, Ulster, September 1989.

[Bem86] T. Bemmerl. Real-time high level debugging in host/target environments. *Microprogramming and Microsystems*, 16:387–400, January 1986.

[BFM+83] F. Baiardi, N. De Francesco, E. Matteoli, S. Stefanini, and G. Vaglini. Development of a debugger for a concurrent language. *SIGPLAN Notices, Symposium on High Level Debugging*, 18(8):98–106, January 1983.

[BFV86] F. Baiardi, N. De Francesco, and G. Vaglini. An interactive debugger for a concurrent language. *IEEE Transactions on Software Engineering*, SE-12(4):547–553, April 1986.

[BG81] R. M. Burstall and J. A. Goguen. An informal introduction to specifications using Clear. In *The Correctness Problem in Computer Science (R. S. Boyer ed.)*, pages 185–214. Academic Press, London, December 1981.

[BGH+89] D. L. Black, D. B. Golub, K. Hauth, A. Tevanian, and R. Sanzi. The Mach exception handling facility. *SIGPLAN Notices, SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):45–56, January 1989.

[BGM89] G. S. Blair, J. J. Gallagher, and J. Malik. Genericity vs inheritance vs delegation vs conformance vs ... *Journal of Object Oriented Programming*, 2(3):11–17, October 1989.

[BH83] B. Bruegge and P. Hibbard. Generalized Path Expressions: A high level debugging mechanism. *SIGPLAN Notices, Symposium on High Level Debugging*, 18(8):34–44, March 1983.

[BH90a] H.-D. Böcker and J. Herczeg. Browsing through program execution. In *Proceedings INTERACT 1990*, pages 991–996. IFIP, July 1990.

[BH90b] H.-D. Böcker and J. Herczeg. TRACK - a trace construction kit. In *Proceedings CHI 1990*, pages 415–422. ACM, April 1990.

[BH90c] H.-D. Böcker and J. Herczeg. What tracers are made of. *SIGPLAN Notices, Proceedings ECOOP/OOPSLA 1990*, 25(10):89–99, October 1990.

[BLW89] C.R. Ball, T. W. Leung, and C. A. Waldspurger. Analysing patterns of message passing. *SIGPLAN Notices*, 24(4):191–193, April 1989.

[Bov86] J. D. Bovey. A debugger for a graphical workstation. *Software Practice and Experience*, 17(9):647–662, September 1986.

[Bri87] E. Brinksma. LOTOS - a formal description technique based on the temporal ordering of observational behaviour. Technical Report ISO/ TC97/ SC21/ 97.21.20.2, International Organization for Standardization, July 1987.

[Bro88] M. H. Brown. Exploring algorithms using Balsa-II. *Computer*, 21(5):14–38, May 1988.

[Bru85] B. Bruegge. *Adaptability and Portability of Symbolic Debuggers.* PhD thesis, Dept. of Comp. Sc. Carnegie-Mellon University, Pittsburgh, Penn., September 1985.

[BS73] A. R. Brown and W. A. Sampson. *Program Debugging, the prevention and cure of program errors.* Macdonald Elsevier, London, September 1973.

[BTM89] A. F. Brindle, R. N. Taylor, and D. F. Martin. A debugger for Ada tasking. *IEEE Transactions on Software Engineering*, SE-15(3):293–304, March 1989.

[BW83] P. Bates and J. C. Wileden. An approach to high-level debugging of distributed systems. *SIGPLAN Notices, Symposium on High Level Debugging*, 18(8):107–111, January 1983.

[Car83a] J. Cardell. Multilingual debugging with the SWAT high-level debugger. *SIGPLAN Notices, Symposium on High Level Debugging*, 18(8):180–189, March 1983.

[Car83b] T. A. Cargill. The Blit debugger. *SIGPLAN Notices, Symposium on High Level Debugging*, 18(8):190–200, March 1983.

[Car86a] J. Carden. Professional debug facility and advanced fullscreen debug. *BYTE*, 11(4):249–255, April 1986.

[Car86b] T. A. Cargill. Pi: A case study in object oriented programming. In *SIGPLAN Notices, Proceedings OOPSLA 1986*, pages 350–360. ACM, November 1986.

[Car89] D. Caromel. Service, asynchrony and wait-by-necessity. *Journal of Object Oriented Programming*, 2(4):12–22, November 1989.

[CB86] W. Cunningham and K. Beck. A diagram for object oriented programs. In *SIGPLAN Notices, Proceedings OOPSLA 1986*, pages 361–367. ACM, November 1986.

[CBM90] W. H. Cheung, J. P. Black, and E. Manning. A framework for distributed debugging. *IEEE Software*, 5(7):106–115, January 1990.

[CC89] E. J. Cameron and D. M. Cohen. The IC* system for debugging parallel programs via interactive monitoring and control. *SIGPLAN Notices, SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):261–267, January 1989.

[CCH+87] A. Carle, K. D. Cooper, R. T. Hood, K. Kennedy, L. Torczon, and S. K. Warren. A practical enviroment for scientific programming. *IEEE Computer*, 8(10):75–89, November 1987.

[Chu83] Y. Chu. High level debugging by interactive direct execution. In *SoftFair 1983*, pages 274–281. IEEE, June 1983.

[Coo86] S. Cook. Languages and object oriented programming. *Software Engineering Journal*, pages 73–80, March 1986.

[Cox86] B. J. Cox. *Object Oriented Programming - An Evolutionary Approach.* Addison Wesley, Reading, Mass., 1986.

[Cox88a] B. J. Cox. *Objective-C Interpreter version 4.0 User's Reference Manual.* StepStone Inc, January 1988.

[Cox88b] B. J. Cox. Outlook. *Journal of Object Oriented Programming*, 1(1):54–57, May 1988.

[Cox90] B. J. Cox. Planning the software industrial revolution - the impact of object-oriented technologies. *IEEE Software*, 7(11):25–33, November 1990.

[CP86] P. Corsini and C.A. Prete. Multibug: Interactive debugging in distributed systems. *IEEE Micro*, 6(3):26–33, June 1986.

[CRS89] E. L. Cusack, S. Rudkin, and C. Smith. An object oriented interpretation of LOTOS. In *FORTE 1989 International Conference on Formal Description Techniques*, pages 265–283, Vancouver, December 1989.

[CS89] D. Callahan and J. Subhlok. Static analysis of low-level synchronization. *SIGPLAN Notices, SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):100–111, January 1989.

[Cus89] E. L. Cusack. Refinement, conformance and inheritance. In *Workshop: the Theory and Practice of Refinement*, pages 1–15, Milton Keynes, January 1989. The Open University.

[CW89] W-H. S. Cheung and V. E. Wallentine. DEBL: A knowledge based language for specifying and debugging distributed programs. *Communications of the ACM*, 32(9):1079–1084, September 1989.

[dB90] F. de Boer. A proof system for the parallel object-oriented language POOL. Technical Report 507, Philips Research Laboratories, Bedrijven, May 1990.

[DC86] A. D. Dewar and J. G. Cleary. Graphical display of complex information within a Prolog debugger. *International Journal of Man-Machine Studies*, 25:503–521, July 1986.

[Deu79] M. Deutsch. Verification and validation. In *Software Engineering*, pages 329–408. Prentice Hall, January 1979.

[DGM88] P. Degano, R. Gorrieri, and S. Marchetti. An exercise in concurrency: A CSP process as a condition/event system. *Lecture Notes in Computer Science*, 340:85–104, January 1988.

[Dij72] E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972.

[Don88] C. Dony. An exception handling system for an object-oriented language. *Lecture Notes in Computer Science, Proceedings ECOOP 1988*, 322:146–161, August 1988.

[Don90] C. Dony. Exception handling and object-oriented programming: Towards a synthesis. *SIGPLAN Notices, Proceedings OOPSLA/ECOOP 1990*, 25(10):322–230, October 1990.

[DP89] A. Doucet and P. Pfeffer. A debugger for O2, an object oriented language. In *TOOLS 1989*, pages 559–571, Nantes, October 1989. SOL Ltd.

[DPW91] C. Dony, J. A. Purchase, and R. L. Winder. Exception handling in object-oriented systems (report on an ECOOP'91 workshop). *OOPS Messenger (in print)*, 1991.

[DS86] N. Delisle and M. Schwartz. A programming environment for CSP. *SIGPLAN Notices*, 22(1):34–41, November 1986.

[Els89] I. J. P. Elshoff. A distributed debugger for Amoeba. *SIGPLAN Notices, SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):1–10, January 1989.

[Fel89] S. I. Feldman. IGOR: A system for program debugging via reversible execution. *SIGPLAN Notices, SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):112–123, January 1989.

[Fid89] C. J. Fidge. Partial orders for parallel debugging. *SIGPLAN Notices, SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):183–194, January 1989.

[FM89] M. B. Feldman and M. L. Moran. Validating a demonstration tool for graphics-assisted debugging of Ada concurrent programs. *IEEE Transactions on Software Engineering*, SE-15(3):305–313, March 1989.

[For89] A. Forin. Debugging of heterogeneous parallel systems. *SIGPLAN Notices, SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):130–140, January 1989.

[Gai85] J. Gait. A debugger for concurrent programs. *Software Practice and Experience*, 15(6):539–554, June 1985.

[GAS+86] P. Garmon, S. Adams, S. Stein, M. Kahl, A. Singer, C. Wang, T. O'Reilly, and J. Strang. *Think's LightSpeedC User's Manual (Macintosh Version)*. Think Technologies, Inc., Bedford, Mass., March 1986.

[GD74] J. D. Gould and P. Drongowski. An exploratory study of computer program debugging. *Human Factors*, 16(3):393–407, June 1974.

[GH88] D. Gelperin and B. Hetzel. The growth of software testing. *Communications of the ACM*, 31(6):687–695, June 1988.

[GKY89] G. S. Goldszmidt, S. Katz, and S. Yemini. Interactive blackbox debugging for concurrent languages. *SIGPLAN Notices, SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):271–282, January 1989.

[Gol83] A. Goldberg. *Smalltalk-80 The Interactive Programming Environment.* Addison Wesley, 1983.

[Gol87] A. Goldberg. Programmer as reader. *IEEE Software*, 4(5):62–70, September 1987.

[Goo75] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80 The Language and its Implementation.* Addison Wesley, 1983.

[Gra83] W. C. Gramlich. Debugging methodology. *SIGPLAN Notices, Symposium on High Level Debugging*, 18(8):1–3, March 1983.

[Ham88] R. Hamlet. Special section on software testing. *Communications of the ACM*, 31(6):662–667, June 1988.

[Har83] M. T. Harandi. Knowledge-based program debugging: A heuristic model. In *SoftFair 1983*, pages 282–288, June 1983.

[HC89] A. A. Hough and J. E. Cuny. Initial experiences with a pattern-oriented parallel debugger. *SIGPLAN Notices, SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):195–205, January 1989.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, August 1977.

[HHG90] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object oriented systems. *SIGPLAN Notices, Proceedings OOPSLA/ECOOP 1990*, 25(10):169–179, October 1990.

[HK89] W. Hseush and G. E. Kaiser. Data Path Debugging: Data-oriented debugging for a concurrent programming language. *SIGPLAN Notices, SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):236–247, January 1989.

[Hoa81] C. A. R. Hoare. The emperor's old clothes. *Communications of the ACM*, 24(2):75–83, February 1981.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[HW89] T. P. Hopkins and M. I. Wolczko. Concurrent programming using Smalltalk-80. *The Computer Journal*, 32(4):341–350, August 1989.

[Int84] S. P. I. International. *VAX/VMS Algol 68 User Guide: VMS DBG.* SPI International, January 1984.

[Joh78] S. C. Johnson. Yacc: Yet Another Compiler-Compiler. Technical Report 32, Bell Laboratories, Murray Hill, New Jersey, July 1978.

[Joh83] M. S. Johnson. Some requirements for architectural support of software debugging. *SIGPLAN Notices, Symposium on High Level Debugging*, 18(8):140–148, March 1983.

[JS85] W. L. Johnson and E. Soloway. PROUST, an automatic debugger for pascal programs. *BYTE*, 8(4):179–190, April 1985.

[Kel88] L. Keller. A guide to testing and debugging. Master's thesis, University College London, London, October 1988.

[Kep87] G. Kepeklian. DTML - user's guide. Technical Report WP2.3.1, Thompson CSF, France, December 1987.

[KG88] M. F. Kleyn and P. C. Gingrich. GraphTrace — understanding object-oriented systems using concurrently animated views. In *SIGPLAN Notices, Proceedings OOPSLA 1988*, pages 191–205. ACM, November 1988.

[KJ88] W. Kozaczynski and A. Jindal. An executable system specification to support the JSD methodology. In *Euro Comp 1988: System Design: Tools and Concepts*, pages 340–349, Brussels, July 1988. IEEE.

[KM91] J. L. Knudson and O. L. Madsen. Exception handling in BETA. Technical report, Aarhus University, Denmark, July 1991.

[KMMPN87] B. B. Kristensen, O. L. Madesen, B. Moller-Pedersen, and K. Nygaard. The BETA programming language. In *Research Directions in Object Oriented Programming (B. D. Shriver and P. Wegner eds.)*. MIT press, January 1987.

[Knu63] D. E. Knuth. Computer-drawn flowcharts. *Communications of the ACM*, 6(9):555–563, September 1963.

[Knu87] J. L. Knudson. Better exception handling in block structed systems. *IEEE Software*, 4(3):40–49, May 1987.

[Koe90] A. Koenig. An exceptional ideal. *Journal of Object Oriented Programming*, 2(3):52–59, July 1990.

[KP86] T. Kaehler and D. Patterson. *A Taste of Smalltalk.* Norton, New York, October 1986.

[KS90] A. Koenig and B. Stroustrup. Exception handling for C++. *Journal of Object Oriented Programming*, 3(2):16–33, July 1990.

[Lac91] S. Lacourte. Exceptions in Guide, an object-oriented language for distributed applications. *Lecture Notes in Computer Science, Proceedings ECOOP 1991*, 512:268–287, July 1991.

[Lam78] L. Lamport. Time, clocks and the ordering of events in distributed systems. *Communications of the ACM*, 21(7), July 1978.

[Lar90] J. R. Larus. Abstract Execution: A technique for efficiently tracing programs. Technical report, Computer Science Dept., University of Wisconsin, Madison, Wisconsin, February 1990.

[Las89] J. Laski. Testing in the program development cycle. *Software Engineering*, pages 95–106, March 1989.

[LD85] R. L. London and R. A. Duisberg. Animating programs using Smalltalk. *Computer*, 18(8):61–71, August 1985.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, Cambridge, Mass., 1986.

[Lis87] B. Liskov. Data abstraction and hierarchy. In *SIGPLAN Notices, Addendum to Proceedings OOPSLA 1987*, pages 17–34. ACM, December 1987.

[LL89] C-C. Lin and R. J. LeBlanc. Event-based debugging of object/action programs. *SIGPLAN Notices, SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):23–34, January 1989.

[LMC87] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computing*, C-36(4):471–482, April 1987.

[LMS85] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Communications of the ACM*, 32(1):52–78, January 1985.

[LMT89] W. R. LaLonde, J. McGugan, and D. Thomas. The real advantages of pure object-oriented systems or why object-oriented extensions to C are doomed to fail. In *Proceedings COMPSAC 1989*, pages 344–350. IEEE, November 1989.

[LS75] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. Technical Report 39, Bell Laboratories, Murray Hill, New Jersey, October 1975.

[LS79] B. Liskov and A. Snyder. Exception handling in CLU. *IEEE Transsctions on Software Engineering*, SE-5(11), November 1979.

[LS80] B. P. Lientz and E. B. Swanson. *Software Maintenance Management.* Addison Wesley, 1980.

[LSAS77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[Mat85] D. C. J. Matthews. Poly manual. Technical report, University of Cambridge, February 1985.

[MC88] B. P. Miller and J. D. Choi. A mechanism for efficient debugging of parallel programs. *SIGPLAN Notices, SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):135–144, June 1988.

[Men87] M. G. Menelaou. Analysis of a questionnaire's responses. Technical Report 2408, University College London, London, June 1987.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall International, May 1988.

[Mey89] B. Meyer. Writing correct software. *Dr Dobbs Journal*, pages 48–63, December 1989.

[MFS90] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–42, December 1990.

[MH89] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computer Surveys*, 21(4):593–622, December 1989.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall Internation Series on Computer Science, London, January 1989.

[MMP89a] O. L. Madsen and B. Moller-Pedersen. Basic principles of the BETA programming language. In *Object Oriented Programming Systems (G. Blair, D. Hutchinson and D. Shepard eds.).* Pitman Publishing, January 1989.

[MMP89b] O.L. Madsen and B. Moller-Predersen. Virtual Classes—a powerful mechanism in object oriented programming. *SIGPLAN Notices, Proceedings OOPSLA 1989*, 24(10), October 1989.

[MMS78] J. G. Mitchell, W. Maybury, and R. Sweet. Mesa language manual. Technical Report 1, Xerox PARC, February 1978.

[Mod79] M. L. Model. *Monitoring System Behaviour in a Complex Computational Environment.* PhD thesis, Palo Alto Research Center, California 94304, January 1979.

[Moh88] T. G. Moher. PROVIDE: A process visualization and debugging environment. *IEEE Transactions on Software Engineering*, SE-14(6):849–857, June 1988.

[MPW89] M. G. Menelaou, J. A. Purchase, and R. L. Winder. On debuggers and debugging: Tools and techniques. Technical Report 89/65, University College London, London, April 1989.

[Mul83] M. A. F. Mullerburg. The role of debugging within software engineering environments. *SIGPLAN Notices, Symposium on High Level Debugging*, 18(8):81–90, March 1983.

[MWPC83] M. A. F. Mullerburg, H. L. Wertz, M. L. Powell, and E. S. Cohen. Integrated environments. *SIGPLAN Notices, Symposium on High Level Debugging*, 18(8):60–62, March 1983.

[Mye79] G. J. Myers. *The Art of Software Testing.* Wiley-Interscience, December 1979.

[Mye83] B. A. Myers. Displaying data structures for interactive debugging. Master's thesis, PARC, 1983.

[Mye84] B. A. Myers. The user interface for Sapphire. *IEEE Computer Graphics and Applications*, 4(12):13–23, December 1984.

[Ner91] J. M. Nerson. Exception handling in Eiffel, July 1991. Presented at ECOOP 1991, W4: Workshop on Object Oriented Exception Handling.

[Neu91] P. G. Neumann. Inside risks: the clock grows at midnight. *Communications of the ACM*, 34(1):170–170, January 1991.

[NH86] V. Nguyen and B. Hailpern. A generalized object model. *SIGPLAN Notices*, 21(10):78–87, October 1986.

[NP90] O. Nierstrasz and M. Papathomas. Viewing objects as patterns of communicating agents. In *SIGPLAN Notices, Proceedings OOPSLA 1990*, pages 38–43. ACM, October 1990.

[OT89] T. O'Reilly and G. Todino. *Managing UUCP and Usenet.* O'Reilly and Associates, California, December 1989.

[PA90] P. C. Philbrow and M. P. Atkinson. Events and exception handling in PS-Algol. *The Computer Journal*, 33(2):108–124, April 1990.

[Pap91] M. Papathomas. A unifying framework for process calculus semantics of concurrent object-based languages and features. In *Object Composition (D. Tsichritzis ed.)*, pages 205–224. Universite De Geneve, Geneva, Switzerland, Universite De Geneve, Geneva, Switzerland, July 1991.

[Pet77] J. L. Peterson. Petri Nets. *ACM Computer Surveys*, 9(3):223–252, September 1977.

[PL86] C. A. Prete and B. Lazzerini. DISDEB: An interactive high level debugging system for a multi microprocessor system. *Microprocessors and Microsystems*, 18:401–408, January 1986.

[PN81] M. C. Pong and N. Ng. PIGS — a system for programming with interactive graphical support. *Software Practice and Experience*, 13:847–855, October 1981.

[Pun90] W. Y. Pun. *A Design Method for Object Oriented Programming.* PhD thesis, University College London, London, September 1990.

[PW89] J. A. Purchase and R. L. Winder. High level debugging of object oriented programs with Message Pattern Specifications. Technical Report 89/76, Univeristy College London, London, October 1989.

[PW90] J. A. Purchase and R. L. Winder. Message Pattern Specifications: A new technique for the detection of errors in parallel object oriented systems. *SIGPLAN Notices, Proceedings OOPSLA/ECOOP 1990*, 25(10):116–125, October 1990.

[PW91a] J. A. Purchase and R. L. Winder. Debugging tools for object-oriented programming. *Journal of Object Oriented Programming*, 4(3):10–27, June 1991.

[PW91b] J. A. Purchase and R. L. Winder. Handling errors in object oriented systems. In *Object Oriented Software Engineering — The Next Step (B. Anderson ed.)*, pages 74–75. BCS OOPS, 1991.

[RdPG91] S. Crespi Reghizzi, G. Galli de Paratesi, and S. Genolini. Definition of reusable concurrent software components. *Lecture Notes in Computer Science, Proceedings ECOOP 1991*, 512:148–166, July 1991.

[Red88] U. S. Reddy. Objects as closures: Abstract semantics of object oriented languages. In *ACM Conference on Functional Programming.* ACM, September 1988.

[Ren82] T. Rentsch. Object oriented programming. *SIGPLAN Notices*, pages 51–57, September 1982.

[Rob90] G. Roberts. CoSIDE design overview. Technical Report UCL-16, Univeristy College London, March 1990.

[RRZ89] R. V. Rubin, L. Rudolph, and D. Zernik. Debugging parallel programs in parallel. *SIGPLAN Notices, SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):216–225, January 1989.

[RSHWW88] G. A. Roberts, J. Sadr-Hashemi, M. Wei, and R. L. Winder. Deliverable 27a: Workpackage 10, design document for the object oriented framework. Technical Report WP10 17, Univeristy College London, London, March 1988.

[RWW88a] G. A. Roberts, R. L. Winder, and M. Wei. Deliverable 27b: Workpackage 10, the sequential prototype of the Solve programming system. Technical Report WP10 22, Univeristy College London, London, December 1988.

[RWW88b] G. A. Roberts, R. L. Winder, and M. Wei. The Solve object oriented programming system for parallel computers. Technical Report WP10 19, University College London, London, October 1988.

[SBK81] S. B. Sheppard, J. W. Bailey, and E. Kruesi. The effects of the symbology and spatial arrangement of software specifications in a debugging task. Technical Report TR-81 388200-4, General Electric, Arlington, Virginia, August 1981.

[SBN89] D. Socha, M. L. Bailey, and D. Notkin. Voyeur: Graphical views of parallel programs. *SIGPLAN Notices, SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):206–215, January 1989.

[Sch71] J. T. Schwartz. An overview of bugs. In *Debugging Techniques in Large Systems (Randell Rustin ed.)*, pages 1–16. Prentice Hall, November 1971.

[Sen83] H. E. Sengler. A model of the understanding of a program and its impact on the design of the programming language grade. *Psychology of Computer Use*, pages 91–106, April 1983.

[Sha82] E. Shapiro. *Algorithmic Programming Debugging.* PhD thesis, MIT Press, July 1982.

[Shn87] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley, London, 1987.

[Shu89] R. N. Shutt. A rigorous development strategy using the OBJ specification language and the MALPAS program analysis tools. In *Proceedings of the 2nd Europian Software Engineering Conference*, pages 260–291. Springer Verlag, September 1989.

[SM88] P. A. Szekely and B. A. Myers. A user interface toolkit based on graphical objects and constraints. In *SIGPLAN Notices, Proceedings OOPSLA 1988*, pages 36–45. ACM, November 1988.

[Smi85] E. T. Smith. A debugger for message-based processes. *Software Practice and Experience*, 15(11):1073–1086, February 1985.

[Som89] I. Sommerville. *Software Engineering.* Addison Wesley, Reading, Mass., September 1989.

[Spi89] J. M. Spivey. *The Z Notation.* Prentice Hall International, 1989.

[ST83] R. Seidner and N. Tindall. Interactive debug requirements. *SIGPLAN Notices, Symposium on High Level Debugging*, 18(8):9–22, March 1983.

[Sta88] R. Stallman. *GDB+ 2.5.0 Manual, the GNU C++ debugger.* GNU Free Software Foundation, Mass., February 1988.

[Sto88] J. M. Stone. Debugging concurrent processes: A case study. In *Proceedings of Workshop on Parallel and Distributed Debugging*, pages 145–153. ACM, June 1988.

[Sto89] J. M. Stone. A graphical representation of concurrent processes. *SIGPLAN Notices, SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1):226–235, January 1989.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison Wesley, Reading, Mass., March 1986.

[Str88] B. Stroustrup. What is object oriented programming? *IEEE Software*, 4(9):10–20, May 1988.

[Thi85] H. Thimbleby. Failure in the technical user-interface design process. *Computers and Graphics*, 9(3):187–193, January 1985.

[TI86a] M. Tokoro and Y. Ishikawa. Concurrent programming in Orient84/K: An object-oriented knowledge representation language. *SIGPLAN Notices*, 21(10):39–48, October 1986.

[TI86b] M. Tokoro and Y. Ishikawa. Orient84/K: A language within multiple paradigms in the object framework. In *Proceedings of the 19th Annual Hawaii Conference on System Sciences*, pages 198–207, November 1986.

[TL91] A. Tiwary and H. M. Levy. Exception handling in concurrent/distributed object oriented environments, July 1991. Presented at ECOOP 1991, W4: Workshop on Object Oriented Exception Handling.

[TR81] T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: A syntax directed programming environment. *Communications of the ACM*, 24(9):563–573, September 1981.

[vdBL89] J. van den Bos and C. Laffra. PROCOL - a parallel object language with protocols. In *SIGPLAN Notices, Proceedings OOPSLA 1989*, pages 95–102. ACM, October 1989.

[Ves86] I. Vessey. Expertise in debugging computer programs: An analysis of the content of verbal protocols. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-16(5):621–637, October 1986.

[Ves89] I. Vessey. Toward a theory of computer program bugs: An empirical test. *International Journal of Man-Machine Studies*, 30(1):23–46, January 1989.

[VJN+90] J. Vitek, B. Junod, O. Nierstrasz, S. Renfer, and C. Werner. Events and Sensors — enhancing the reusability of objects. In *Object Management (D. Tsichritzis ed.)*, pages 345–356. University of Geneva, Geneva, Switzerland, July 1990.

[vT74] D. van Tassel. *Program Style, Design, Efficiency, Debugging and Testing.* Prentice Hall, New York, December 1974.

[vV89] T. van Vleck. Three questions about each bug you find. *Software Engineering Notes*, 14(5):62–63, July 1989.

[Weg90] P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, August 1990.

[Wei82] M. Weiser. Programmers use Slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[Wer82] H. Wertz. Stereotyped Program Debugging: An aid to novice programmers. *International Journal of Man-Machine Studies*, 16:379–392, January 1982.

[Wik87] Å. Wikström. *Functional Programming Using Standard ML.* Prentice Hall International, 1987.

[Wil85] S. Williams. *Programming the 68000.* Sybex, 1985.

[WN88] R. L. Winder and J. Nicholson. JDB: An adaptable interface for debugging. *Software Practice and Experience*, 18(3):221–238, March 1988.

[Wol88] M. Wolczko. *Semantics of Object-Oriented Languages.* PhD thesis, University of Manchester, Manchester, September 1988.

[WP88] R. L. Winder and W. Y. Pun. The Smalltalk-80 browser: A critique. Technical report, Univeristy College London, London, March 1988.

[YB85] S. Yemini and D. M. Berry. A modular verifiable exception handling mechanism. *TOPLAS*, 7(2):214–243, April 1985.

[YC79] E. Yourdon and L. L. Constantine. *Structured Design.* Prentice Hall, October 1979.

[Yeh77] R. Yeh. *Current Trends In Programming Methodology Vol 2: Program Validation.* Prentice Hall, November 1977.

[Yel89] P. M. Yelland. First steps towards fully abstract semantics for object oriented languages. In *Proceedings ECOOP 1989*, pages 347–364, Nottingham, July 1989. Oxford University Press.

[YT87] Y. Yokote and M. Tokoro. Concurrent programming in ConcurrentSmalltalk. In *Object-Oriented Concurrent Programming (Y. Yokote and M. Tokoro eds.)*, pages 129–158. MIT Press, 1987.