Message Pattern Specifications: A New Technique for Handling Errors in Parallel Object Oriented Systems

Jan A. Purchase & Russel L. Winder
Department of Computer Science, University College London
Gower Street, London WC1E 6BT.

ABSTRACT

As object oriented techniques enable the fabrication of ever more sophisticated systems, the need grows for a mechanism to ensure the consistent and 'correct' behaviour of each object at run-time. We describe a new, in-source specification mechanism, Message Pattern Specifications (MPS), to directly satisfy this need in a succinct, orthogonal and disciplined manner. Targeted for use in parallel object oriented systems, MPS allows programmers to enunciate the 'legal' patterns of run-time behaviour in which their objects may engage. Furthermore, it supports the definition of methods for object recovery or graceful failure in case these specifications are violated during execution.

1 INTRODUCTION

Despite the common assertion that program reliability is one of the most significant problems remaining in software engineering, it has been observed that remarkably few object oriented languages cater, in any pragmatic manner, for the detection and handling of anomalous behaviour at run-time [Mey89, Mey88, LG86]. The increase in language sophistication heralded by the object oriented paradigm has, so far, failed to produce a commensurate increase in the facility of error detection mechanisms [Mey89] or debugging tools [PW89b].

In this paper, we present a technique to enhance the reliability of software at the level of individual objects. Our goal is to demonstrate the feasibility of operational specifications as the basis of a technique for in-source specification and run-time error handling. We describe MPS — Message Pattern Specification — an in-language facility for the specification of object behaviour and the definition of relevant recovery procedures, should these behaviours be violated. MPS augments a 'host' object oriented language and allows the user to express the desired conduct of each defined object in terms of one or more Message Patterns, using a language based on CSP [Hoa83]. A group of *shadow methods* is also defined by the user, in the host language, to handle the recovery or graceful failure of objects which violate one of their message pattern specifications.

This work is divided into four main parts. In the next section, we describe the failings of the few error handling systems that currently exist for object oriented languages and explain the advantages of operational specification — upon which MPS is based. Sections 3 and 4 contain a detailed definition of the theory and usage of Method Pattern Specifications; including the need for *shadow methods* to handle MPS violations. MPSs have been implemented within the object oriented language Solve [RWW88a]. This implementation, and examples of how MPSs are used within an object oriented language, are discussed in section 5. We conclude with a comparison of our work to the facilities offered by alternative techniques and a discussion of further research.

2 TOWARDS ACCOUNTABLE OBJECTS

2.1 TRADITIONAL METHODS

In order to enhance program reliability, it has been the tradition to embed tests within the main methods of objects, designed to check for erroneous usage and invalid states. This practice can be a liability in sophisticated objects, as these tests — *error signallers* (ESs) — can be both numerous and complex. Often, they obscure the semantics of the methods containing them, rendering code harder to understand, debug or maintain [Ben87]. In many cases the code segments used to alleviate signalled problems — the *error handlers* (EHs) — are also placed within the main method body, exacerbating the situation.

This problem has been partially overcome in a few languages through the direct, in-language, support of ESs and EHs. These facilities are collectively known as *error signaller and handler mechanisms* (ESHMs). Typically, these express assertions about the current system state and nominate handlers to be executed if these are violated. Such mechanisms are still in their infancy and suffer from four principle problems:

- **Weakness of Expression**: Many ESHMs, e.g., those of Eiffel [Mey88] and EC++ [Mas89], use the host language to specify these assertions. Although this greatly eases the use of such mechanisms, it also compromises their power. Object oriented languages are not the most expedient tool for behavioural specification, especially in concurrent systems.

- **Non-Uniform Error Handling**: In order to reduce complexity, assertion violations should be signalled and handled orthogonally, irrespective of where they originate. Languages with weak or *retro-fitted* ESHMs, for example Smalltalk-80 [GR83] and C++ [Str86], ignore this point and use different techniques to process exceptions originating from hardware, the operating system and the user's program. In C++, there is little commonality between the way in which hardware and user software exceptions are handled; in Smalltalk-80 the latter are not even provided.

- **Lack of Implicit Discipline**: Some ESHMs override all existing control structures of the host language. These can, and are, abused as a means of easy escape from heavily nested constructs to remote 'saviour' procedures. Such mechanisms leave too much to the discretion of the user, rendering them open to ill conceived and inconsistent use. The Ada *raise* mechanism and associated ESHM exemplify this [Mey88].

- **Poor Distribution**: Many ESHMs, for example that of CLU [LG86], make little distinction between ESs, EHs and primary method code, allowing all three to be mixed at will. As noted previously, this can severely obscure method semantics.

Object accountability requires an unambiguous division of responsibility, within a program, for 'normal' behaviour of an object and the detection and management of anomalies [BGH$^+$89]. The latter requires language support for the axiomatic definition of 'correct' object behaviour and a uniform mechanism to process all exceptions without contravening the control constructs of the host language.

2.2 THE BENEFITS OF OPERATIONAL SPECIFICATION

The need to enunciate the correct behaviour of an object implies the use of a specification language. However attempts to use an established specification language like 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 run-time behaviours. Ideally, a means of expressing behaviour in a full predicate calculus should be provided. Alas, this too is beyond the realm of current pragmatism [Mey88, KJ88]. Indeed, any method of specification which concentrates on internal form and semantics is likely to be non-viable, because of the dearth of popular, mathematically rigourous, parallel object-oriented languages. Instead, we use a method based on object behaviour.

Operational specifications are expressed solely in terms of the events suffered by objects. Object behaviour, in a parallel system, is specified by a partial ordering of instances of these events. The power of this method is chiefly responsible for the wealth of event-based debugging and monitoring techniques for parallel and distributed systems [Bat87, 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 slicing [Wei82], a vital part of behavioural analysis. Furthermore, such models are entirely language independent and support concurrency with relative ease.

3 MESSAGE PATTERN SPECIFICATIONS

3.1 PRIMITIVE ELEMENTS 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, the authors have derived this fundamental alphabet [PW90]. Implemented in CSP, the model is operationally isomorphic to a object oriented system (as defined by [BGM89]) with parallelism to method granularity, shared memory, one-way communication and a by-proxy delegation mechanism. 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. Each event class and the attributes (parameters) which distinguish individual instances are described below.

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

- **Object Assignment.** The event class *assign*, parameterized by the instances involved.

- **Object Destruction.** The event class *destroy*, parameterized by the instance destroyed.

- **Message Send.** The event class *send*, parameterized by sender, recipient, selector and arguments.

- **Method Starts Execution.** The event class *execute*, parameterized by method (method name and process identifier) and the host instance.

- **Method Terminates Execution.** The event class *terminate*, parameterized by method (method name and process identifier) and the host instance.

- **Delegation (or superclass lookup).** The event class *lookup*. Occurs when a selector from an incoming message fails to match any method in the local dictionary, so it is forwarded to the proxy. It is parameterized by original recipient, selector, arguments and proxy (or superclass).

- **Delegation (or Inheritance) Path Terminates.** The event class *lfail*. Occurs as above, except that no further proxy can be provided or the root of the inheritance hierarchy has been reached during a search. It is parameterized by recipient, selector and arguments.

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

3.2 MPS SYNTAX

MPS serves to fully, or partially, specify the desired operational behaviour of objects at run-time. They form part of the definition of every class or type template expressed in the host object oriented language. An MPS is said to *fail* if tr, the trace of all events actually exhibited by an object, does not conform to the behaviour pattern the MPS describes. Each MPS consists of five parts:

- **A Name.** All specifications are labelled with user-defined names to enhance their meaning. Names are essential in identifying which (if any) specifications fail at run-time.

- **The Relevant Trace.** The subtrace of tr pertaining to the specification; 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. Partial specifications can use wildcard traces to express their partiality. These wildcards are named, bound variables. If tr conforms to the event pattern, these variables become instantiated with those components of the actual behaviour trace to which they correspond.

- **Additional Constraints.** These are optional conditions which need to be met to ensure that a specification is satisfied. Expressed in a language similar to the specification clauses of CSP [Hoa83], these clauses often express further provisos in terms of the bound variables described above.

- **Handler.** This is the name of the local, private method (shadow method) which is used to handle the error generated if a specification is violated (see section 4). If none is provided, a default is used.

Operator Syntax	*Semantics*	
	CSP	*English*
tr restrict ℓ	$tr \upharpoonright \ell$	trace tr filtered by those events (or event classes) in the list ℓ
reverse tr	$\overline{tr}$	the order of trace tr is reversed
tail tr	tr'	the first element of tr is removed

Table 1: Functions Used to Generate Relevant Traces

The overall MPS syntax is:

```
         Name
satisfies [relevant-trace
inwhich   template
iff       constraints]
else      handler
```

Some elucidation on the these constituents follows.

3.3 THE RELEVANT TRACE

The relevant trace, the first part of an MPS, filters from tr these aspects of behaviour which are of interest to a particular specification. Filtration is an essential feature of event monitoring systems [Bat89], which enhances their ability to specify system behaviour at many different levels of abstraction. Filtration avoids the consideration of irrelevant events, improving the efficiency of man and machine. Relevant traces are functions of tr designed to yield sub-traces which 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. Filtering is achieved though use of the *tr restrict* ℓ construct, which restricts the relevant trace to events (or event types) contained within the list ℓ. This function has semantics identical to the CSP operator $\upharpoonright$. Other operators include those listed in Table 1.

For example, the relevant trace (for a stack object)

$$tail\ (tr\ restrict\ \{execute(push),\ execute(pop)\})$$

is a stream of events representing all *push* and *pop* operations on the stack, excepting the first. Whereas

$$tr\ restrict\ \{execute\}$$

is a stream of all events of type *execute*.

3.4 THE SPECIFICATION TEMPLATE

The main body of an MPS is a template, constructed from instances of the primitive event classes using pattern operators. It

Operator Syntax	*Semantics*	
	CSP	*English*
$\rho*$	$\mu X.(\rho \rightarrow X)$	the pattern ρ occurs an unspecified number of times in succession
$\rho * n$	ρ^n where $(n \in \mathbb{N}_1)$	n successive instances of the pattern ρ occur
$\bar{}\rho$	$x : (\alpha tr - \rho) \rightarrow \ldots$	any pattern *except* ρ
ρ_1, ρ_2	$\rho_1 \rightarrow \rho_2$	pattern ρ_2 occurs immediately after ρ_1
$\rho_1 \ldots \rho_2$	$\rho_1 \rightarrow X \backslash \{\rho_2\}; \rho_2$	pattern ρ_2 occurs after ρ_1
$\rho_1 \backslash / \rho_2$	$(\rho_1 \rightarrow X) \square (\rho_2 \rightarrow X)$	pattern ρ_1 or ρ_2 occurs
$\rho_1 \Vert \rho_2$	$\rho_1 \vert\vert\vert \rho_2$	the patterns ρ_1 and ρ_2 are interleaved
$\rho/[\beta]$	$\rho \nleftarrow \beta$	as the pattern ρ occurs, the state β is attained

Table 2: Pattern Operators Used to Generate Specifications Templates

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 handle parallel and state compositions. Also, unlike LEX, the specification is designed to aid semantic review. To help users to quickly determine how an object is used and its limitations. These functions are vitally important to software reuse [SBK81].

Each pattern operator takes one or more events or patterns (ρ) and yields a more complex pattern. The operators are listed in Table 2, wherein the process denoted X is a *don't care* term, $\mathbb{N}_1$ is the set of all *strictly positive* numbers and the state expression β is a boolean condition binding the observable state of the object concerned. Encapsulation must not be violated in verifying state conditions.

The simplest pattern is merely a single event specification. For example $terminate(push)$, which signifies the termination of the method *push*. The MPS language supports the specification of sequential and concurrent sequences (,, ..., ||), deterministic and non-deterministic choice ($\backslash/$) and iteration ($*$, $*n$) to compose useful behaviour patterns from these events.

As an example: for an object representing a screen, a specification to ensure that, in a redraw method, a screen is cleared before the flood of redraw messages is sent, might read:

$$\begin{array}{ll} FinishClear & \\ satisfies & [tr\ restrict\{send,\ execute,\ terminate\} \\ inwhich & execute(redraw), execute(wipe), \\ & terminate(wipe)...(send(redraw))*] \end{array}$$

In many specifications, there is a need to label events to distinguish instances of the same event occurring within partially concurrent, *overlapping* threads. This is achieved by labelling processes with symbolic names which are unified with a process identifier at run-time. Such labelling is accomplished through use of the : operator. Hence to indicate a sequence of events a, b and c, in which a and c are generated by the same thread i, one may use the template:

$$i : a,\ j : b,\ i : c$$

Here j may, or may not equal i. Both are instances of *unified variables*. For example, to enforce serialisation of the *push* method (i.e., to monitor it) one would use a specification:

$$\begin{array}{ll} StackMonitor & \\ satisfies & [tr\ restrict\ \{execute,\ terminate\} \\ inwhich & i : execute(push), i : terminate(push)] \end{array}$$

to ensure that both events belonged to the same thread.

3.5 ADDITIONAL CONSTRAINTS

Relevant traces and specification templates are not enough to describe all behaviours. 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 the operators ... and || cannot be constrained in any way as part of the specification — a further limitation.

To circumnavigate these weaknesses, an optional clause can be used to supplement some MPSs. Any partially specified sections of the template, upon which further constraint should be imposed, and which hitherto had been represented with ... or ||, is instead denoted by a bound trace variable. These variables are denoted by a name, preceded by the symbol \$ (e.g., \$$a$). They are instantiated with the traces they represent at runtime, if tr conforms to the template. The constraint clause may then strengthen the MPS by expressing additional conditions for acceptance in terms of these traces, their lengths and their alphabets. Table 3 depicts the functions that constraints use, in addition to these shown in Table 1, to manipulate these trace variables or calculate their properties. Another class of bound variable is used to achieve an analogous unification of values within state expressions. These variables are denoted by names prefixed by the symbol &.

The traces and alphabets generated by constraint functions (through the use of bound trace variables or 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 $\#$), state bound & variables (found using state expressions) and unified variables can be likewise compared using standard magnitude relations ($>$, $>=$, etc). The semantics of any relation may be negated by prefixing it with the symbol ˜.

Operator	*Semantics*	
Syntax	*CSP*	*English*
head *tr*	tr_0	the first element of trace *tr*
$tr[n]$	$tr[n]$ where ($n \in \mathbb{N}_1$)	the n^{th} element of trace *tr*
$tr[n..m]$	$\wedge_{n \le i \le m} tr[i]$ where ($n, m \in \mathbb{N}_1$)	a subtrace a *tr* consisting of the n^{th} through to the m^{th} element of *tr*
$@tr$	αtr	the alphabet of trace *tr*
$\#tr$	$\#tr$	the length of trace *tr*
$tr!\ell$	$\#(tr \uparrow \ell)$	the number of occurrences of ℓ within *tr*
$tr_x / \backslash\, tr_y$	$tr_x{}^\wedge tr_y$	trace (or event) concatenation

Table 3: MPS Constraint Functions

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 MPS:

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

although this is not the best method, it alleviates the need for a precondition check in this case. Assuming the existence of a method *size*, another specification achieving the same goal is:

```
NotEmpty
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]). Postconditions may likewise be simulated in MPS, e.g., to ensure *push* increases stack size:

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

Often, additional constraints are used as their name implies. For example, for a window object to assert that it should only be moved when opened and destroyed when closed, one may specify:

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

4 RUN-TIME ERROR MANAGEMENT

4.1 UNIFORMITY OF ERROR HANDLING

In a system supporting MPS, each instantiated specification constitutes a *guardian* parser for its object, constantly monitoring events relevant to itself and the object it guards. At run-time, an MPS may have one of three states: *satisfied*, indicating behavioural compliance of the object to its specification; *violated*, indicating that the object has failed to act as specified and *undecided*, indicating that, as yet, insufficient information is available for it to attain one of the former states. A satisfied specification is ignored by the system, the MPS is reset and continues parsing anew. A failed specification, however, causes the MPS to abort parsing and signal an error condition. This results in the execution of the handler method. The responsibility of the MPS error signaller ends here.

The error handling method, a *shadow method* named in the last (optional) argument of a MPS, is passed the name of the failing specification and a full description of the event trace causing the violation. It may to programmed to take one of the following courses of action:

- **Attempt Recovery.** The EH could alter the state of the object to undo any damage caused by the erroneous behaviour, alter the state of the object to avoid further instances of the error, rewind the parser by one pattern and attempt to continue execution.
- **Graceful Failure.** The EH may alter the state of an object in order to render it usable by other threads, create diagnostics sufficient to describe the nature of the failure to the user (see section 7) and advise the object's clients of the problem by violating the client's implicit *serverOK* specification (see below).
- **Invoke the System Debugger.** This behaviour is only acceptable at the early stages of software development. It is especially useful if the debugging tool also supports MPS (see section 4.2)

Each object has an implicit MPS:

```
ServerOK
satisifes   [tr
inwhich     ~serverfail]
```

When any handler adopts the graceful failure policy, its final action is to automatically place an instance of the special event class *serverfail* (and a reference to itself as a parameter) into the input stream of all of its client's parsers. This results in the failure of the client's *serverOK* MPS. By default (unless the user redefines this MPS), all *serverOK* violations result in graceful failure. In this manner, propagation of error though the compositional hierarchy of a system is achieved.

The facility and uniformity of MPSs is greatly enhanced by virtue of the fact that common hardware or operating system exceptions (e.g., divide by zero, memory allocation problems) may be handled as pre-defined primitive MPSs. The core set of event classes defined in section 3.1 may be extended to encompass occurrences of these errors. Allowing the user, or library programmer to make specifications like:

```
NoZeroDivide
satisfies      [tr
inwhich        ~DivideByZero]
else           handler
```

4.2 LINKS WITH OBJECT-ORIENTED DEBUGGING

The ability of MPSs to report deviations from desired behaviour in parallel object oriented systems constitutes a useful facility for a debugger. Indeed, MPS was originally designed as a tool for testing behavioural hypotheses during debugging [PW89a]. Its mode of usage within an interactive debugging tool is somewhat different to that described here. However, MPS demonstrates considerable aptitude at both preventing bugs and debugging.

5 IMPLEMENTATION

5.1 THE SOLVE LANGUAGE

The MPS system is being implemented as part of SOLVE, the Span Object oriented Language enVironmEnt. At the core of SOLVE is Solve, an object oriented language developed at University College London and designed for use on parallel, multiprocessor systems [RWW88a, RSHWW88, RWW88b]. It supports fully active objects with concurrently executing methods. Furthermore, it provides a message passing subsystem which allows applications running outside Solve, perhaps in different languages, to inter-communicate. This communication can be achieved synchronously, asynchronously or with futures. Solve supports multiple inheritance, type conformance and parameterized types.

At compilation, Solve source is parsed and a parse tree of node objects is created. These are instances of node classes, of which one exists for each of Solve's constructs. These nodes respond to messages to symbol check, type check and code generate themselves. Hence the latter stages of compilation are achieved by sending these messages in turn to the parse tree root node, which propagates them accordingly. The compiler supports incremental compilation to the level of individual language statements and since all primitive methods are user defined, it can generate code in any intermediate-level, object oriented language capable of supporting its abstractions.

Many aspects of the Solve language and system can be fundamentally altered by the user, for example, the inheritance lookup mechanism, scheduling mechanism and binding mechanism. The implementation of MPS takes full advantage of this flexibility.

5.2 EFFICIENT SUPPORT OF MPSs IN SOLVE

The implementation of MPS comprised three main tasks: altering the SOLVE compiler to correctly parse MPS definitions; altering the code generator to include the instrumentation needed for the system to log fundamental events and building the guardian parsers for each object.

The first of these tasks necessitated editing the LEX and YACC [LS75, Joh78] grammars of the original compiler and creating new parse tree node classes to support MPS type checking and code generation. The Solve language syntax was extended in a manner as to ensure backward compatibility with earlier, non-MPS, versions. The MPS parse tree nodes themselves contain references to filter, template and constraint nodes which construct relevant trace filters, guardian parsers and constraint provers respectively. Finally, object nodes were amended to enable receipt of event streams.

Instrumentation, the amendment of code such that it has side effects which make visible its internal behaviour, is notoriously computationally expensive [Bat89]. Consequently, it must be implemented efficiently and be subject to user veto. To support this, the code generator was designed to generate instrumentation only when the relevant compiler options are selected. If these options are deselected, MPSs are disabled and have *no* runtime overhead. Each event type requires unique instrumentation. The *create* and *destroy* events classes can be provided by instrumenting the memory management system; *assign* by careful alteration of the assignment parse tree node; *send* by changing the communication subsystem; *execute* and *terminate* by amending the scheduler and *lookup* and *lfail* through the provision of a new binding algorithm. Each instrumentation is implemented as an 'in-line' function call in the generated code.

In section 3.4, MPSs are compared to LEX specifications. This comparison is a limited one however. Because MPSs allow the parallel composition of event sequences, they cannot be directly implemented by LALR(1) parsers. Instead, a finite state automaton is required which recognises a symbol hierarchy rather than a flat alphabet. This allows an executing automaton to delegate responsibility for parsing a complex event sequence to a subordinate automaton (which it regards as a symbol). Constrained Shuffle Automata (CSA)[Bat87] satisfy all of these criteria. In addition, they permit event instances to be selectively ignored according to the values of their attributes, thus providing support for filtration and additional constraints. Efficient implementations of constrained shuffle automata have been reported elsewhere [Bat89], however our implementation is not yet mature enough to test.

```
Signature Datafile(element)
Supertypes (Object)

InstanceOperations
  openout: ()-><Datafile(element)>
  openin:  ()-><Datafile(element)>
  read:    ()-><element>
  write:   (<element>)-><Datafile(element)>
  close:   ()-><Datafile(element)>

TypeOperations
  new:  (<String>, <Type>)-><Datafile(element)>
```

Figure 1: Solve Signature for the Datafile Type

5.3 MPS EXAMPLE

The following example is provided in order to give some indication of how MPSs are used in practice. The example chosen is a simple one, with the aim of conveying pertinent facts without undue complication. The *datafile* type describes a set of objects which may be used to simplify the use of file storage in a system. Each instance of datafile represents a file. The Solve signature of this parameterized type is defined in figure 1. The type has five instance methods: *openin* and *openout*, which open files for reading and writing respectively; and *read*, *write* and *close*, which have the usual file semantics. All methods are parameterless except *write*, which requires a datum to append to the file. All methods return a modified datafile object except *read*, which returns a datum from the file. The type method *new* creates new instances of datafiles, given the pathname and type of data stored (or to be stored) therein.

The Solve implementation of Datafile consists of three main constituents:

1. **Definitions of Instance and Type Methods**
2. **Method Pattern Specifications**
3. **Definitions of Handler Methods**

Constituents 1 and 3 are largely irrelevant to this example, in that they are Solve specific and have no bearing on MPS. However, in order to convey some flavour of the Solve language, the beginning of the implementation is depicted in figure 2. Datafile objects have two instance variables: *storage*, an *IOBuffer* which actually performs the disc storage; and *pathname*, a string which holds the name of the file.

Figure 3 depicts a list of the MPSs constraining Datafile behaviour. Two MPSs are defined, each of which triggers a different handler when violated (*FileUseError* and *PermError*). These handlers are Solve methods, defined in a similar way to those of figure 2. Their exact usage is beyond of the scope of this work. The MPS *FileProtocol* ensures that a file may have either multiple readers or one writer, but not both concurrently.

```
Implementation Datafile(element)
Includes (Object)

InstanceSection
  let storage <IOBuffer(element)> := ()
  let pathname <String> := "."

  export const read <Method((), <element>)> :=
    [ let next <element> := storage<--get()
    =>next
    ]

  export const write <Method((<element>),
      <Datafile(element)>)> :=
    [ item <element> |
    storage<--put(item)
    ]

...
```

Figure 2: Solve Implementation of the Datafile Type

It uses a unified variable n (see section 3.4) to ensure that each reader closes the datafile after use. The interleaving operator $||$ ensures that file accesses need not be nested. The MPS *AccessPermission* ensures that, within a given process, file access permissions are not infringed. It is violated, for example, if a file opened with *openin* attempts to execute the method *write*. The same MPS prevents closed files from being reclosed.

6 RELATED WORK

The parallel language PROCOL [vdBL89] offers facilities that, like MPS, may be used to order and constrain inter-object communications and to control object access. These tools can be used to specify some of the behaviours we have previously described (iteration, sequence and selection), both sequentially and concurrently. However, they fail to incorporate inheritance or delegation into their model (a crucial facet of object oriented languages [BGM89, Weg87]). Subsequently, since it constrains usage protocols by receiver, it must name specific object instances within a specification — an unfortunate inflexibility. Furthermore, PROCOL's tools offer no support for the assignment event class (proved essential by [Bru85]). In PROCOL, event specifications are mandatory and may *not* be partially specified. PROCOL fails to handle possible run-time errors indicated by failure of one of its specifications, instead the thread concerned is suspended until it can satisfy the behaviour pattern. This, coupled with the provision of parallelism to object and not method granularity, means that recursive methods may deadlock the system.

MPSs' template facility bears some resemblance to the *lifescript* of Kozaczynski's Executable System Specification for JSD (ESSJSD)[KJ88]. Although not aimed specifically at object oriented languages — indeed, much research has shown that JSD is an inappropriate method for object-oriented design

```
MPSSection

FileProtocol
satisfies [ tr restrict
                 {execute(openin),
                 execute(openout), execute(close)}
inwhich    create;
           ((execute(openin);
           (execute(openin)*n||execute(close)*n);
           execute(close))
           \/ (execute(openout); execute(close)))*;
           destroy ]
else       FileUseError

AccessPermission
satisfies [ tr restrict i:execute
inwhich    (execute(openin);
           execute(read)*; execute(close))
           \/ (execute(openout);
           execute(write)*; execute(close)) ]
else       PermError
```

Figure 3: Message Pattern Specifications of the Datafile Type

[Som89, Pun90] — ESSJSD's goals have some commonality to those of MPSs. ESSJSD offers no direct support for inheritance, delegation or partial specifications. Furthermore, ESSJSD is a design aid — it does not pass into the implementation and has to be regressed if the latter is subject to alteration during maintenance. ESSJSD's ESs are, by the authors' admission, very verbose and are not separated from the main specification in any way. ESSJSD's specifications are expressed in Prolog and then translated, implying some reliance on the logic paradigm. Finally, it is our view that specification languages should be engineered to support a software engineering paradigm not just one design methodology.

The Data Path Debugging (DPD)[HK89] technique shares, with MPS, the ability to monitor object behaviour as a stream of events and to report deviations of actual traces from these specified. However, MPS is a preventative technique — object behaviour is specified initially so that deviations may be noticed and acted on immediately. However, DPD is a debugging tool for locating errors once their presence has been revealed by testing. MPSs constitute part of a program, they aid semantic review, whereas the specifications of DPD are ephemeral. DPD and MPS have some commonality however, both systems avoid language dependencies. Both use standardized multiple history graphs to log the events suffered by object instances and then allow the comparison of desired and actual behaviour. However, we feel that the DPD technique puts too strong an emphasis on the state changes of objects, to the detriment of its handling of control flow. Whilst it is true that many bugs are incarnated in inappropriate data values, the root cause is often a flawed flow of control — both aspects of the dichotomy must be addressed. Rather than attempting to build a unified fundamental event set for control and state, as we have, DPD builds one and attempts to map it to the other — an activity which is not always feasible [LL89]. Many of DPD's facilities support the behavioural *patching* of software, we feel that with the advent of modern incremental compilers, this dangerous practice is no longer necessary.

7 FURTHER WORK

We are currently implementing MPS and anticipate that performance will be the chief problem. The cheapness of processes in the SOLVE system causes the additional overhead due to instrumentation and CSA maintenance to be significant. In a parallel system, the perturbation of method timing due to instrumentation — the *probe effect* [Gai85] — can alter program behaviour and is to be avoided at all costs. The authors are currently seeking an optimum implementation of constrained shuffle automata.

Currently, each MPS can have only one failure mode, because of the problems of deducing the semantics underlying a specification violation, after it has occurred. We plan to overcome this limitation by allowing the expected points of failure to be described as part of the MPS template. Furthermore, it may be useful to allow each failure point to designate a separate handler.

Often the most pragmatic error handlers simply report the problem and fail gracefully. Currently, these reports are little more than formatted listings of the failed specification name, the event causing the failure and its attributes. The listings often start with the root cause and backtrack through the compositional hierarchy because of the $serverOK$ propagation explained in section 4.1. A detailed textual analysis of the departure of actual behaviour from specified behaviour, or a visual trace of the reasons for failure, would be more useful. The authors are currently investigating the latter.

The event classes listed in section 3.1 pertain to a *pure* object oriented model, in which all data entities are objects. However, many popular object oriented languages are based on a *hybrid* object model (e.g., C++ [Str86]), in which two data representations are used. Simple data entities, like integers and reals, are represented as primitive types and lack any form of object interface or message passing protocol. More complex entities are built from these and constitute objects, identical to those of the of the pure model. If MPSs are to support this hybrid paradigm, research must be conducted to establish if, and how, the fundamental event set of the hybrid object model differs from that of the pure model.

8 CONCLUSION

We have developed an original means of operational, in-source specification of the behaviour of objects in a parallel, object oriented language. These specifications allow the user to define, detect and handle cases where the run-time behaviour of a system deviates from that which the user intended. They allow programmers to enhance program reliability without degrading readability. Indeed, they improve the latter because they separate conventional code from error signalling and handling code, and allow the objects to openly declare 'legal' usage patterns. The MPS technique permits behavioural specification at many levels of abstraction due to its ability to filter events and its use of a template (and constraint) language based on CSP. It constitutes a systematic, language independent, orthogonal exception handling mechanism and contributes directly toward object accountability.

ACKNOWLEDGEMENTS

This work has been partially funded by a quota award studentship, courtesy of the Science and Engineering Research Council.

References

[Bat87] 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. In *SIGPLAN Workshop on Parallel Debugging*, pages 11–22, New York, January 1989. ACM.

[Ben87] J. Bentley. *Programming Pearls*. Addison Wesley, Reading, Mass., 1987.

[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. In *SIGPLAN Workshop on Parallel Debugging*, pages 45–56, New York, January 1989. ACM.

[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. *Symposium on High Level Debugging in SIGPLAN Notices*, 18(8):34–44, March 1983.

[BLW89] C.R. Ball, T. W. Leung, and C. A. Waldspurger. Analysing patterns of message passing. *SIGPLAN Notices*, 24(4):191–193, April 1989.

[Bru85] B. Bruegge. *Adaptability and Portability of Symbolic Debuggers*. PhD thesis, Dept. of Comp. Sc. Carnegie-Mellon University, Pittsburgh, Penn., September 1985.

[Gai85] J. Gait. A debugger for concurrent programs. *Software Practice and Experience*, 15(6):539–554, June 1985.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80 The Language and its Implementation*. Addison Wesley, 1983.

[HK89] W. Hseush and G. E. Kaiser. Data path debugging: Data-oriented debugging for a concurrent programming language. In *SIGPLAN Workshop on Parallel Debugging*, pages 236–247, New York, January 1989. ACM.

[Hoa83] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, New York, 1983.

[Joh78] S. C. Johnson. Yacc: Yet another compiler-compiler. Technical Report 32, Bell Laboratories, Murray Hill, New Jersey, July 1978.

[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.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. The MIT Press, Cambridge, Mass., 1986.

[LL89] C-C. Lin and R. J. LeBlanc. Event-based debugging of object/action programs. In *SIGPLAN Workshop on Parallel Debugging*, pages 23–34, New York, January 1989. ACM.

[LS75] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. Technical Report 39, Bell Laboratories, Murray Hill, New Jersey, October 1975.

[Mas89] G. Masotti. *A Tutorial Introduction to EC++: Extended C++*. University of Southern California, Los Angeles, CA, September 1989.

[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.

[Pun90] W. Pun. *A Design Method for Object Oriented Programming*. PhD thesis, University College London, London, March 1990.

[PW89a] J. A. Purchase and R. L. Winder. High level debugging of object oriented programs with message pattern specifications. Research Note 89/76, Univeristy College London, London, October 1989.

[PW89b] J. A. Purchase and R. L. Winder. Object oriented debugging tools. Research Note 89/77, Univeristy College London, London, October 1989.

[PW90] J. A. Purchase and R. L. Winder. A model for concurrent object oriented systems based on communicating sequential processes. Research Note 90/38, University College London, London, July 1990.

[RSHWW88] G. A. Roberts, J. Sadr-Hashemi, M. Wei, and R. L. Winder. Deliverable 27a: Workpackage 10, Design document for the object oriented framework. Research Note 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. Research Note 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. Research Note 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.

[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.

[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.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, Reading, Mass., March 1986.

[vdBL89] J. van den Bos and C. Laffra. Procol - a parallel object language with protocols. In *OOPSLA 1989*, pages 95–102. ACM, October 1989.

[Weg87] P. Wegner. Dimensions of object-based language design. In *OOPSLA 1987*. ACM, October 1987.

[Wei82] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.