DIVISION OF COMPUTER SCIENCE

A Review of Twenty Formal Specification Notations

R J Vinter

Technical Report No. 240

February 1996

A Review of Twenty Formal Specification Notations

R J Vinter

Division of Computer Science,
University of Hertfordshire.

Tel: +44-01707-285122
Email: R.Vinter@herts.ac.uk

February 1996

Abstract

This report describes a general review that was conducted as part of the Human Cognition and Formal Methods research project at the University of Hertfordshire during the latter half of 1995. The main purpose of the survey was to determine which of the formal notations currently available would be most suitable as a model for testing the project's specific research theories. Twenty notations from the state-based, process algebra and algebraic domains of formal specification were selected and reviewed against a list of predefined criteria. This report discusses the review's findings and discusses some of the main intellectual problems facing the designers of formal notations.

1 Introduction

A software specification is an abstract description of a proposed system's functional and non-functional behaviour intended to serve as the basis for its design or implementation. A formal specification is one that is written entirely in a formal notation, that is, "a language with an explicitly defined syntax and semantics" [LIS79, page 277]. Most formal systems, or "formalisms", contain a formal notation and a deductive apparatus [WOO88]. The formal notation component is normally based around on or more systems of formal logic, which provide a basis for describing in precise, mathematical terms the software systems to be developed. The deductive apparatus component, comprising axioms and inference rules, provides an independent means by which designers can test whether their intuitive theories about selected properties of a specification are correct.

Although formal methods have been available for over thirty years, industry has generally been reluctant to adopt them because of doubts surrounding their commercial viability. This might explain designers' preoccupation with natural language based techniques in the past, which has tended to result in the creation of large, unwieldy documents that are prone to ambiguity, inconsistency and error. However, gradual advances in formal methods technology and some highly successful applications in industry have helped to dispel many of the software engineering community's initial doubts and formal methods are now gradually gaining widespread acceptance. In fact, it is widely believed that formal logic based notations could hold the key to overcoming some of the classical problems associated with program specification. However, the number of formal notations has been steadily increasing as a result of ongoing academic research and the types of formalism now available vary widely in their purpose, popularity and logical foundations. Hence, the main purpose of this review was to determine which particular notation would be most likely to fulfill the specific requirements of the Human Cognition and Formal Methods project; not to critically compare each formalism's effectiveness for solving particular types of software engineering problem.

2 Types of Formalism

2.1 The State Based Approach

Under the state based (or "model-oriented") approach to formal specification, an abstract representation of a system is constructed using well defined underlying calculi (such as propositional or predicate logic) and common mathematical tools (such as sets, relations or functions). However, the formalisms that fall under this category often contain slight extensions or variations of these. System behaviour is expressed by specifying operations that cause transitions in local or global states - these operations are typically constructed using standard data types. Examples of state based formalisms include Gypsy, VDM and Z.

2.2 The Process Algebra Approach

Under the process algebra approach, a system is described in terms of individual processes which may intercommunicate and interact with their environment. Generally, the formalisms that fall under this category tend to be aimed at the specification of systems in which processes may run in parallel and intercommunicate, such as networked or concurrent applications. Most process algebras allow designers to specify a logical and temporal order for operations which tends to give rise to detailed specifications of system behaviour. Examples of this approach include the formalisms CCS, CSP and LOTOS.

2.3 The Algebraic Approach

Algebraic (or "equational") specification techniques aim to describe objects in terms of relationships between the operations that use them. Typically, these objects are specified as abstract data types such as arrays, lists or queues. Algebraic specifications contain no notion of machine state and do not describe system behaviour explicitly, at least in the same sense as those produced under the state based approach. Under the algebraic approach, a program is modelled as a "many-sorted algebra", which comprises sets of values and operations on those sets; each set is given a name, called its "sort". The "signature" of a data type comprises its sorts, the names of its operations, and the sorts of these operations' domains and ranges. An "abstract data type" is a class of many-sorted algebras with the same signature and some specified common properties. Hence, an "algebraic data type" is the definition of an abstract data type by means of a signature and some axioms (i.e. logical formulae) that the algebrae of the class must satisfy. The nature of algebraic notations make them particularly well suited to applications such as the specification of programming language semantics. Some popular formalisms that fall under the algebraic approach include Clear, Larch and OBJ.

3 A Review of Twenty Formal Notations

The Human Cognition and Formal Methods project plans to conduct a series of psychological reasoning experiments within the grammatical framework provided by a formal notation. It also plans to formulate complexity metrics based on the findings from such experiments. It is therefore important that the notation chosen to fulfill this role is both popular and that its grammatical foundations are representative of those found in other formal notations. A methodological approach was adopted to the review of each formalism, guided mainly by the project's requirement for specific types of information. In particular, the following information was sought.

- *Founder.* The institution or personnel responsible for the initial creation of the notation and those responsible for any major subsequent developments.
- *Date Founded.* The approximate year in which the notation was first developed.
- *Underlying Calculi.* A description of the notation's logical foundations.
- *Class of Notation.* A statement of the class under which the notation may be categorised - i.e. state-based, process algebra or algebraic.
- *Target Application Type.* The kind of software system that the notation is particularly suited to describing, if applicable.
- *References.* Recommended sources of further reading.
- *Popularity.* An indication of the extent to which the notation has gained acceptance in academia and industry.
- *Examples.* A concise specification (or a subset thereof) expressed in the notation is presented in order to give a general idea of the grammatical constructs and symbology belonging to the notation.

3.1 ADL

The Algebraic Design Language (ADL) is a higher-order specification language developed by Kieburtz and his colleagues at the Oregon Graduate Institute of Science and Technology. Having been developed during the early 1990s, it represents one of the more recent of the formalisms reviewed. Although the language has yet to be fully developed, a preliminary definition of the main core of the ADL notation is given by Kieburtz and Lewis [KIE94]. The semantics of the ADL notation are based entirely on equational logic and set theory. Modularity is achieved through the use of "functors," which are conceptually isolatable units abstracted with respect to ADL structure algebras and coalgebras. Each signature declaration specifies a variety[1] of structure algebras, the terms of a particular algebra, and the free term algebra of the signature - the latter, according to Kieburtz and Lewis, corresponds to a data type in a typed, functional programming language, such as ML or Miranda.

In order to specify a program using algebras, one must first decide upon appropriate control and data structures. Control is expressed through the use of type-parametric combinators applied to inductively defined algebras. ADL's data type definitions are always inductive and construct types from a finite number of alternatives, each of which is a typed N-tuple. ADL includes strong type-checking features. All well-typed ADL programs are guaranteed to terminate, although, checking the correctness of ADL expressions can sometimes give rise to proof obligations which cannot be discharged automatically, that is, using machine verification. The syntax of ADL bears close resemblance to that of the ML language and, like Standard ML, ADL contains a core language with a modular structure. The following example shows a specification for a binary tree "divide-and-conquer" algorithm.

```
signature Btree{type c; btree(a)/c = {$emptytree, $node of c * a * c}}
sort = hom[btree] Btree{c := list(int);
                        $emptytree := Nil,
                        $node := \(xs,x,ys) append xs (Cons(x,ys))}
              (\xs case xs of
                        Nil => $emptytree
                        | Cons(x,xs') =>
                                  let (ys,ys') = part x xs'
                                  in $node(ys,x,ys')
                        end)
```

Adapted from Kieburtz and Lewis [KIE94].

[1] Here, the term "variety" is used to refer to a class of algebras which share a common signature.

One of the fundamental aims underlying ADL's development was ease of program refinement - i.e. "the improvement of algorithms by meaning-preserving, algebraic transformation of programs" [KIE94]. The notation's basis upon equational logic supports this aim by making ADL amenable to program transformation according to the equational theories of its algebras. Owing to its relatively recent development, both industrial and academic application of ADL is still quite limited. According to its designers, one intended use of the language is to provide the computational semantics of specialised software design languages.

3.2 Affirm

Affirm provides an integrated, interactive system for the specification and verification of software systems. It was first developed at the University of Southern California during the early 1980s in order to test the application of algebraic specification research theories propounded by the likes of Guttag and Horning [GUT78] to complex applications. The Affirm formalism includes an axiomatic specification language based on the theory of abstract data types, a verification condition (VC) generator, an interactive theorem prover for verifying properties of specifications or programs, and a library of predefined data types - queues and sets, for example. An Affirm specification contains a series of data type definitions and a number of equational axioms describing the semantics of each type. By presenting a specification in such a manner, it is possible to achieve a high level of abstraction and so avoid low level implementation details. Each Affirm data type definition may comprise three sections, described as follows:

- *Constructors.* Operations which create values of the data type being defined, and whose range is restricted to that same data type.
- *Extenders (or modifiers).* Operations which also have the data type being specified as their range, but unlike constructors, they are not needed to express values of the data type - that is, these operations can be expressed in terms of constructors.
- *Selectors (or predicates).* Operations which yield values with types other than the one being specified.

Adapted from Sunshine et al. [SUN82].

The following example illustrates the Affirm notation. It shows an excerpt from a simple message buffering system in which a single message may be communicated from sender to receiver simultaneously. A sender must wait until the corresponding receiver has processed the previous message before the next one can be sent.

```
type SimpleMessageSystem;
needs types Message, QueueOfMessage, ControlState;
declare s: SimpleMessageSystem;
declare m: Message;
interface State(s): ControlState;
interfaces Sent(s), Received(s), Buffer(s): QueueOfMessage;
interfaces InitializeService, UserSend(s,m), SendComplete(s),
   UserReceive(s), ReceiveComplete(s): SimpleMessageSystem;
interface Induction(s): Boolean;
axioms
   State(UserSend(s,m)) == if State(s) = ReadyToSend
                           then Sending
                           else State(s),
   State(SendComplete(s)) == if State(s) = Sending
                             then ReadyToReceive
                             else State(s),
```

```
    State(UserReceive(s)) == if State(s) = ReadyToReceive
                                 then Acking
                                 else State(s),
    State(ReceiveComplete(s)) == if State(s) = ReadyToReceive
                                     then Acking
                                     else State(s),
    State(InitializeService) == ReadyToSend;
nochange State, Sent, Received, Buffer;
end {SimpleMessageSystem};
```

Sunshine, Thompson, Erickson, Gerhart, and Schwabe [SUN82].

Affirm's natural deduction theorem prover enables designers to prove properties about data types interactively - these properties are expressed in the predicate calculus in the form of theorems. Affirm automatically simplifies propositions using the axioms associated with data types (as rewrite rules). These axioms must be proved as theorems of each data type's definition using the theorem prover. Following this first stage of verification, each data type is normally implemented as a file of separate routines in a programming language so that the VC generator can then be applied to the implementation and its results proved using Affirm's theorem prover [COH86].

Owing to its heavy reliance upon equational axioms to describe the operational semantics for its specifications, the Affirm notation may be classified as an algebraic specification technique. Although the notation was developed primarily for the purpose of specifying abstract data types, it is also ideally suited to the description of concepts underlying state transition machines. It is particularly amenable to describing events that occur in protocol systems, the behaviour of processes and communication channels. However, Affirm has been found to contain a number of limitations. In particular, Sunshine et al. [SUN82] emphasise its lack of support for specifying and verifying concurrent processes, and handling and recovering from error conditions. Nevertheless, the Affirm formalism has been influential in the design of the Larch notation.

3.3 Anna

The Anna notation and its accompanying tool set were developed by the Program Analysis and Verification group at Stanford University during 1980s. Anna ("Annotated Ada") adopts an altogether different approach to the formal specification process. Its designers came to the conclusion that it would not be possible to develop a perfect specification language in a single "giant leap" because the current state of software knowledge did not fully comprehend all of the component activities of the development process at the time of the notation's design. Thus, Anna's designers decided to adopt an evolutionary approach. Rather than designing a new specification language from first principles, they believed that it would be wiser in the long term to experiment with primitive specification notations initially. They believed that it was only through gaining insight and experience from the use of these that they could begin to develop more powerful ones. Anna represents one such primitive language.

Conventionally, whilst informal text annotations tend to aid a human reader's comprehension of a program, they do not help a machine to validate the program's behaviour. However, Anna's designers believed that, by expressing such comments in a formal language, they can be translated into executable form and used to verify a program's behaviour. This is the central philosophy behind the design of the Anna language. Thus, Anna's designers began with the target programming language itself (i.e. Ada) and simply extended it with features that are useful for

specifying and verifying program behaviour. In fact, many of Anna's constructs are simply extensions of Ada's original features. These constructs enable designers to annotate various Ada entities such as: packages, generic units, exceptions, types, statements and assertions. So, an Anna program is effectively an Ada program containing formal, Boolean-valued annotations. Annotations are commonly referred to as "formal comments" and are preceded by the "-- |" symbol. An annotation may refer to any entity of the Ada program that is visible within the annotation's scope. Anna reserved words may be used to bind an annotation to a particular Ada statement or else define its application in some way. Annotations can, for example, be used to specify constraints on an entity or else to specify its interrelations with other entities in the program. The following examples illustrate the basic types of annotation possible.

1. *A type annotation:*

```
     type SQUARE is record
          X, Y : NATURAL;
     end record
-- |      where S : SQUARE => S.X = S.Y;
```

2. *A result annotation of a function:*

```
     function SQUARE(X : NATURAL) return NATURAL;
-- |      where return X * X;
```

3. *An* **out** *annotation of a procedure:*

```
     procedure QUICKSORT(A : in out VECTOR);
-- |      where out (A = SORTED(in A));
```

4. *An object annotation:*

```
     S : SQUARE;
-- | S.X > 0 and S.Y > 0;
```

5. *An assertion:*

```
     S := (3, 3);
-- | SQUARE(S.X) = 9;
```

Luckham [LUC90, pages 20-21].

According to Luckham [LUC90], there are four main ways of using annotations within the Anna language. These are described as follows:

1. *Specification of programs prior to implementation.*
   Annotations are used to express formally the behaviour and properties of a program before it is implemented.
2. *Goal-oriented programming.*
   Annotations are used to define the goals of each major program component.
3. *Definition of run-time checks.*
   Checks may either be used temporarily (for debugging purposes) or be incorporated permanently into a program.
4. *Documentation.*
   All annotations provide a form of formal documentation for the underlying program once it is completed.

Adapted from Luckham [LUC90, pages 10-11].

One particularly useful application of annotations during program development is to act as a guarantee to the users of a package that subprograms in the package will, when implemented, possess the stated properties. An annotation may, therefore, be a constraint on a package body to implement these properties. However, it should be noted that annotations, like normal Ada program statements, are subject to certain computational hazards, such as exception propagation. In addition to annotations, Anna supports a secondary type of comment called "virtual text," which is preceded by the "-- :" symbol. All virtual text must conform to valid Ada syntax and semantic rules just as if it were part of the underlying program. However, virtual text does not affect computation of the underlying program in any way. It is included for a number of reasons: to aid debugging or tracing, to define "virtual attributes" (which can be referred to in annotations and virtual text), and to provide a means for naming or parameterising expressions. The Anna notation also incorporates some of the concepts from the predicate calculus, such as quantified and conditional expressions. Furthermore, it permits the use of some common mathematical symbols including: **not**, **and**, **or**, $\rightarrow$, $\leftrightarrow$, **for all**, **exist**, $/=$, $<=$ and $>=$. These have been included mainly in order to encourage the use of abstraction - i.e. to express functional properties without stating computational details. The following specification illustrates a first (informal) attempt at specifying a "stack" data type and its associated operations in the Anna notation.

Informal stack specification.

```
generic
      type ITEM is private;
      MAX : POSITIVE;

package STACK is

      - Basic functions.
-- : function MEMBER(X : ITEM) return BOOLEAN;
-- : function LENGTH return NATURAL;
-- : function TOP return ITEM;

      OVERFLOW, UNDERFLOW : exception

      - Actual operations.
      procedure PUSH(X : in ITEM);
-- | where
      - If the stack length is MAX then propagate OVERFLOW,
      - otherwise increment the stack length and put the item on top.

      procedure POP(Y : out ITEM);
      where
      - If the stack is empty then propagate UNDERFLOW, delete
      - its top item and store it in Y.

-- | axiom
      - All stacks are initially empty, PUSH adds an item and does
      - not delete items, and PUSH and POP have inverse effects on
      - the STACK.

end STACK;
```

Adapted from Luckham [LUC90, pages 174-175].

The following example illustrates how program development under the Anna framework progresses; the informal, English annotations in the original specification have been translated into formal Anna annotations.

Formal stack specification.

```
generic
      type ITEM is private;
      MAX : POSITIVE;

package STACK is

      -- Basic functions.
--  : function MEMBER(X : ITEM) return BOOLEAN;
--  : function LENGTH return NATURAL;
--  : function TOP return ITEM;
--  : function "=" (S, T : STACK'TYPE) return BOOLEAN;

      OVERFLOW, UNDERFLOW : exception

      -- Actual operations.
      procedure PUSH(X : in ITEM);
--  | where in STACK.LENGTH = MAX => raise OVERFLOW,
--  |       out (STACK.LENGTH = in STACK.LENGTH+1),
--  |       out (STACK.TOP = X);

      procedure POP(Y : out ITEM);
--  | where in STACK.LENGTH = 0 => raise UNDERFLOW,
--  |       out (STACK.LENGTH = in STACK.LENGTH-1),
--  |       out (Y = in STACK.TOP);

--  | axiom
--  | for all S : STACK'TYPE; U, V : ITEM =>
--  |       STACK'INITIAL.LENGTH = 0,
--  |       not STACK'INITIAL.MEMBER(U),
--  |       S[PUSH(V)].MEMBER(U) = S.MEMBER(U) or U = V,
--  |       S[PUSH(U); POP(V)] = S;

end STACK;
```

Adapted from Luckham [LUC90, pages 175-176].

Amongst Anna's tool set is an Annotation Transformer module, which translates annotations into run-time checks, and a Verifier module, which analyses a program in order to determine whether it will behave consistently with its annotation. Other available tools include a run-time checking tool and a symbolic execution tool. However, before Anna can be applied to complex, large-scale development projects with any degree of success, Luckham believes that it must undergo a number of significant improvements. Anna's current limitations are listed as follows:

1. All programs specified in Anna must eventually be implemented in the Ada programming language.
2. There are currently no special annotations for Ada's tasking constructs.[2]
3. Anna's current facilities for manipulating annotations are severely limited.
4. The Anna constructs that define abstract interfaces between Ada programs contain deficiencies that cannot be corrected simply by adding annotations.

[2] According to Luckham, this is due to the fact that research has failed to determine the kinds of annotation that would be useful for the process of specifying concurrent systems.

3.4 The B-Method

The B-Technologies are a suite of tools, notations and methods designed to facilitate the practical application of formal methods to complex software development projects. Its three major components are: the B-Method, the B-Tool, and the B-Toolkit. Their development began at BP International and Oxford University's Programming Research Group during the mid-1980s, and further research and development continued at B-Core from 1993. Under the B-Method, specifications, designs and implementations are presented as "Abstract Machines". An Abstract Machine is a software component comprising a state together with operations on that state, and is specified using Abstract Machine Notation (AMN). An AMN specification contains three conceptually separable parts. Firstly, there is a set of global declarations, such as constraints, constants and abstract sets. Secondly, there is a set of abstract states expressed using set-theoretic concepts, such as sets, relations, functions and sequences. Finally, there is a set of operations on those abstract states, which are modelled using pre- and post-conditions, and constructs for guiding functionality, such as sequencing and choice. AMN also includes a variety of features aimed at promoting the structuring of specifications. These include facilities for manipulating specification modules, such as: importation, inclusion, using, refining, parameterisation, renaming, extending and hiding.

All of AMN's constructs are defined under the Generalised Substitution Language (GSL). In fact, it is an underlying set of syntactic rewriting rules, which substitute AMN expressions for corresponding lower level GSL terms, that form the basis for AMN's "pseudo-programming" specification notation. GSL is based on Dijkstra's guarded command notation[3] but includes extensions for unbounded choice, pre- and post-conditioning. The following examples illustrate how AMN constructs can be defined in the GSL calculus:

AMN Substitution	GSL Substitution
`BEGIN` S `END`	S
`IF` P `THEN` S `ELSE` T `END`	$P \Rightarrow S [] \neg P \Rightarrow T$
`ANY` z `WHERE` P `THEN` S `END`	$@z.(P \Rightarrow S)$

Adapted from B-Core [BCO95a].

According to B-Core [BCO95b], the B-Method is actually based on a number of embedded notations, where AMN represents the highest level notation. These embedded notations include: a logical notation, basic set notation, a relational notation, a mathematical object notation, the GSL notation and the AMN notation. Significantly, since the expression of an Abstract Machine's implementation is based on a set-theoretical model and its operations are described using a restricted subset of AMN, a designer can use virtually the same notation for the purposes of both specification and implementation. The following example of the AMN notation specifies the simple arithmetic operations: "Clear," "inc" and "decrease".

```
MACHINE
   Simple
VARIABES
   number
INVARIANT
   number : NAT
OPERATIONS
   Clear =
      BEGIN number := 0 END;
```

[3] E.W. Dijkstra, *A Discipline of Programming.* Prentice-Hall, 1976.

```
    inc =
        BEGIN number := number + 1 END;
    decrease =
        PRE
            number > 0
        THEN
        ANY newnum WHERE
            newnum < number &
            newnum : NAT
        THEN
            number := newnum
        END
    END
END
```

B-Core [BCO95a].

According to B-Core [BCO95a], the B-Theory Language (B-TL) is a programming language that is specifically intended for the development of software engineering tools like generators, translators and verifiers. A B-TL program comprises a series of theories, each containing a number of rules and some data (i.e. formulae) which the program may manipulate. "Tactics" are used to determine exactly when a specific rule is applied. In fact, specifications, designs and implementations written in AMN are data upon which B-TL programs operate. The B-Tool language interpreter and run-time environment provides a means for checking and executing B-TL programs.

The B-Toolkit is a set of integrated tools which use the B-Tool as a form of software platform and is designed to support the B-Method's integrated software development environment. This contains: a syntax-checker and type-checker for AMN, a proof obligation generator, interactive and automatic theorem-proving assistants, an automated C code generator, a specification animator, an automated documentation preparation tool, a library of reusable specified and coded modules, a proof printer, an interface generator and a configuration management system. The B-Toolkit thus caters for all aspects of the software engineering process from specification through to implementation and maintenance. Furthermore, it encourages an object-oriented approach and the reuse of specified and coded modules.

One of its designers' main aims was to ensure that the B-Method promotes incremental development. This was achieved by providing it with facilities for program modules to be specified and verified in isolation. Another of its designers' aims was to ensure that specified or coded modules are reusable. This was achieved by allowing for Abstract Machines to be parameterised so that instances of machines could be created and reused. Owing to the B-Method's reliance on machine states and its use of set-theoretic concepts to model such concepts, its underlying notation clearly falls under the category of model-based specification techniques. Despite its relatively recent release, the B-Method has already gained popular acceptance in industry and academia. It has been applied on a number of large scale industrial projects and is also currently being taught on numerous computer science degree programmes across the country.

3.5 CCS

Milner's Calculus of Communicating Systems (CCS) provides a process algebra[4] notation and a range of facilities for manipulating and reasoning about expressions

[4]Milner prefers to classify CCS as a "process calculus" rather than a "process algebra" because the former is a more general term implying the inclusion of traditional forms of logic and branches of mathematics [MIL89, page 4].

in the notation. It was developed during the 1970s and, like Hoare's CSP notation, is based around the notion of indivisible interactions between processes, or "agents". An agent may perform an action itself independently or else intercommunicate with other agents - where a communication between two agents does not pass any data value, this is called a "synchronisation". The CCS process algebra incorporates a considerably extended set theory and its own "synchronous" calculus.[5] Expressions in the calculus may be constructed in five main ways: Prefix, Summation, Composition, Restriction and Relabelling. The symbolism of the language is exemplified in the following "Jobshop" specification in which there exists four agents and various actions involving them.

$$
\begin{aligned}
\mathit{Jobshop} &\widehat{=} (\mathit{Jobber} \mid \mathit{Jobber} \mid \mathit{Hammer}) \setminus \\
&\quad \{\mathrm{geth}, \mathrm{puth}\} \mid \mathit{Mallet}) \setminus \{\mathrm{getm}, \mathrm{putm}\} \\
\\
\mathit{Hammer} &\widehat{=} \mathrm{geth}.\mathit{Busyhammer} \\
\mathit{Busyhammer} &\widehat{=} \mathrm{puth}.\mathit{Hammer} \\
\mathit{Mallet} &\widehat{=} \mathrm{getm}.\mathit{Busymallet} \\
\mathit{Busymallet} &\widehat{=} \mathrm{putm}.\mathit{Mallet} \\
\mathit{Jobber} &\widehat{=} \mathrm{in}(\mathit{job}).\mathit{Start}(\mathit{job}) \\
\mathit{Start}(\mathit{job}) &\widehat{=} \textbf{if } \mathit{easy}(\mathit{job}) \textbf{ then } \mathit{Finish}(\mathit{job}) \\
&\quad \textbf{else if } \mathit{hard}(\mathit{job}) \textbf{ then } \mathit{Usehammer}(\mathit{job}) \\
&\quad \textbf{else } \mathit{Usetool}(\mathit{job}) \\
\mathit{Usetool}(\mathit{job}) &\widehat{=} \mathit{Usehammer}(\mathit{job}) + \mathit{Usemallet}(\mathit{job}) \\
\mathit{Usehammer}(\mathit{job}) &\widehat{=} \overline{\mathrm{geth}}.\overline{\mathrm{puth}}.\mathit{Finish}(\mathit{job}) \\
\mathit{Usemallet}(\mathit{job}) &\widehat{=} \overline{\mathrm{getm}}.\overline{\mathrm{putm}}.\mathit{Finish}(\mathit{job}) \\
\mathit{Finish}(\mathit{job}) &\widehat{=} \overline{\mathrm{out}}(\mathit{done}(\mathit{job})).\mathit{Jobber}
\end{aligned}
$$

Adapted from Milner [MIL89, pages 27-31].

CCS also provides a set of proof tools for reasoning about specifications. Milner's Labelled Transition System consists of a set of inference rules designed to check the validity of transitions between states in CCS expressions. Equational and Static laws may be used to determine the behavioural equality of two agents. An Expansion law is used for expanding the behaviour of agents expressed as restricted compositions. CCS also provides various mechanisms that enable one to determine the precise degree of equivalence that holds between two systems.[6]

Unlike model-based specifications, process algebras such as CCS and CSP generate explicit rather than implicit specifications and appear well suited to the specification of concurrent systems. Both represent the execution of a process as a sequence of events, where the execution of each event consumes no time whatsoever. Thus, any two independent processes running concurrently are commonly referred to as an "interleaving of events". According to Hull, the main differences between CCS and CSP lie in the various ways in which they can manipulate the specification of a proposed system [HUL93]. For instance, CSP provides a unique facility for specifying allowable traces of events formally and explicitly. These are the properties of requirements that a system must satisfy. The notation also provides facilities

[5] A fundamental principle of the synchronous calculus is that processes running concurrently always perform at least one action per time period, despite the fact that this might not always be possible in reality.

[6] For details of CCS's notions of equivalence, the reader is referred to Milner's chapters on Strong Equivalence, Observation Equivalence and Observation Congruence in *Communication and Concurrency* [MIL89].

for reasoning about these traces - they can, for example, be represented as lists and their members as elements of sets. CCS, on the other hand, contains no such equivalent facility. Another striking difference is the inclusion of an internal handshaking operation τ ("tau") within CCS. A τ action is invisible to external agents but becomes significant when contemplating the possible forms of equivalence that can hold between two systems in CCS.

3.6 CIRCAL

CIRCAL (CIRcuit CALculus) is a process algebra which can be used to describe, verify and simulate concurrent systems. These systems may be hardware or software based, although, in fact all previous industrial applications of CIRCAL have been confined to the specification and verification of digital hardware. The notation was developed by Milne [MIL85] during the early 1980s with the main aim being to create a calculus with considerable expressive power yet conceptual simplicity. CIRCAL is based on a combination of Milne's dot calculus[7] and set theory, but contains a number of modifications and extensions. One such extension is that CIRCAL permits the representation of simultaneously occurring distinct actions, whereas the dot calculus, CCS and CSP each enforce the arbitrary interleaving of concurrent actions. This makes it possible to describe synchronous and asynchronous behaviour, and allows for the representation and analysis of timing properties at different levels of detail. The design concepts central to CIRCAL, and its differences from other process algebras, are encompassed within six "core" operators: guarding, choice, nondeterminism, termination, concurrent composition and abstraction.

1. *Guarding* $((\alpha\beta)P)$. Guarding appends a term to a label-set creating a new term.
2. *Choice* $(P+Q)$. The choice operator produces a composite term which may perform different actions - in this case, those of P or those of Q.
3. *Nondeterminism* $(P \oplus Q)$. Actions belonging to P or Q may occur but the environment has no control as to whether P or Q is chosen.
4. *Termination* $(P\Delta)$. The termination operator represents "no further action". In this case, only the P action is performed and then the agent terminates.
5. *Concurrent composition* $(P \bullet Q)$. In CCS, communication is represented by a synchronisation between at most two agents. However, concurrent composition in CIRCAL allows synchronised multiway interaction - i.e. where one sender may interact with two or more receivers.
6. *Abstraction* $(-\alpha)$. The CIRCAL abstraction operator allows systems to be described at various levels of information content.

   Adapted from Milne [MIL85].

Like most process algebras, the behaviour of a CIRCAL agent is described in terms of the various interactions that it may have with the other agents in its environment. The "term" construct is used to represent such behaviour. The set of ports through which an agent may interact is called its "sort". Multiple agents may have the same sort, although, those that utilise these ports differently do not possess the same semantic behaviour. The set of sorts belonging to terms describe the spatial properties of a system, whilst the terms themselves express its behaviour. Like all sorted algebras, every CIRCAL operator is defined to have a particular sort.

A form of operational semantics, called "acceptance semantics," is used to provide CIRCAL with its facilities for active experimentation. These give meaning

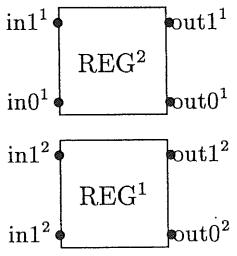
[7] G.J. Milne, *Abstraction and nondeterminism in concurrent systems.* In *Proceedings of the Third International Conference on Distributed Computing Systems*, IEEE Computer Society Press, 1982.

to CIRCAL's primitive operators and form the basis from which to construct and execute proofs about system behaviour.[8] Milne argues that acceptance semantics capture our intuitive understandings of the behaviour of concurrent systems. CCS's observational semantics restricts a designer to merely observing actions, whereas CIRCAL permits active experimentation. Its semantics enable users to determine how agents, represented by terms, would respond to a stimulus on any of its ports or connectors. Milne states that two possible kinds of result may arise from an experimentation: either the stimulus will be accepted with the term evolving to a new term or else the stimulus will be rejected with the metasymbol "$*$" being produced [MIL85]. This is expressed formally as follows:

> PROG$_L$ is the set of all terms of sort L. For terms T, $T' \in$ PROG$_L$ and label-set $m \subseteq L$, the relation $<< T, m >, T' >$ for sort L is written as
>
> $$T \xrightarrow{m} T' \text{ for } T,\ T' \text{ of sort } L.$$
>
> Here term T accepts the m stimulus and evolves to T'.
>
> If $T \xrightarrow{m} *$ then m is not supported by T, and the result of an m experiment on T is the special symbol $*$. For instance, if term T is defined in such a way as to exclude interaction on its α port, the experimenting on its α port causes "state" $*$ to be reached, $T \xrightarrow{\alpha} *$, and T *rejects* α.
>
> Adapted from Milne [MIL85].

Active experimentation provides a basis for conducting equivalence proofs in CIRCAL. Two terms P and Q may be considered equivalent (written $P \approx Q$) if they produce equivalent terms given the same stimulus for all stimuli taken from the powerset of their sort - although, P and Q must have the same sort. For any given stimulus, if "$*$" results, then experimentation on an equivalent term must also generate "$*$" for the two terms to be considered equivalent [MIL85].

CIRCAL includes an "abstraction operator" which enables systems to be modelled at different levels of detail and allows for the introduction of nondeterminism. Nondeterminism appears when we are unable to predict or control the behaviour of one of a number of events. But Milne believes that nondeterminism is always resolvable at a certain level of detail - i.e. some agent is always capable of resolving the choice as to how events occur. Thus, the extent to which nondeterminism manifests itself depends largely upon the amount of detail that is abstracted away. The following example illustrates how the CIRCAL notation can be used to model precisely an n-bit binary register memory device.



[8] Acceptance semantics are used by Milne in [MIL85], for example, to underpin a system of proof properties by establishing a set of consistent CIRCAL laws including: idempotency, commutativity, associativity and identity.

A (MEM^n) memory device comprises n registers, defined as follows:

$\mathrm{REG}^1\ \mathrm{REG}^2 \bullet \cdots \bullet \mathrm{REG}^n$.

Each register has its own input and output lines:

$\mathrm{REG}^i \Longleftarrow \mathrm{REG}\{\mathrm{in0}^i/\mathrm{in0}\}\{\mathrm{in1}^i/\mathrm{in1}\}\ \{\mathrm{out0}^i/\mathrm{out0}\}\{\mathrm{out1}^i/\mathrm{out1}\}$

Thus, a 2-cell memory device is specified by:

$$
\begin{array}{ll}
\mathrm{REG}^1 \Longleftarrow & \mathrm{in1}^1\ \mathrm{REG}^1(1) + \mathrm{in0}^1\ \mathrm{REG}^1(0) \\
\mathrm{REG}^1(1) \Longleftarrow & \mathrm{out1}^1\ \mathrm{REG}^1(1) + \\
 & \mathrm{in0}^1\ \mathrm{REG}^1(0) + \mathrm{in1}^1\ \mathrm{REG}^1(1) + \\
 & (\mathrm{out1}^1\ \mathrm{in0}^1)\ \mathrm{REG}^1(0) + \\
 & (\mathrm{out1}^1\ \mathrm{in1}^1)\ \mathrm{REG}^1(1)
\end{array}
$$

$$
\begin{array}{ll}
\mathrm{REG}^1 \bullet \mathit{REG}^2 = & \mathrm{in}^1[\mathrm{REG}^1(1) \bullet \mathit{REG}^2] \\
 & + \mathrm{in0}^1[\mathrm{REG}^1(0) \bullet \mathit{REG}^2] \\
 & + \mathrm{in1}^2[\mathrm{REG}^1 \bullet \mathit{REG}^2(1)] \\
 & + \mathrm{in0}^2[\mathrm{REG}^1 \bullet \mathit{REG}^2(0)] \\
 & + (\mathrm{in1}^1\ \mathrm{in1}^2)[\mathrm{REG}^1(1) \bullet \mathit{REG}^2(1)] \\
 & + (\mathrm{in0}^1\ \mathrm{in0}^2)[\mathrm{REG}^1(0) \bullet \mathit{REG}^2(0)] \\
 & + (\mathrm{in1}^1\ \mathrm{in0}^2)[\mathrm{REG}^1(1) \bullet \mathit{REG}^2(0)] \\
 & + (\mathrm{in0}^1\ \mathrm{in1}^2)[\mathrm{REG}^1(0) \bullet \mathit{REG}^2(1)]
\end{array}
$$

Adapted from Milne [MIL85].

3.7 COLD-1

Development of COLD-1 (Common Object-oriented Language for Design) began in the early 1980s when Jonkers[9] first began work combining properties of algebraic and state based specification techniques. The notation was later developed at Phillips Research in Eindhoven, during the late 1980s, where it has been used in the development of various consumer electronics devices. It is still being used as a basis for ongoing language research at Phillips Research Laboratories. COLD-1 is a wide-spectrum language which aims to cater for program specification, design and implementation. Its underlying logical foundations have been extended to include first-order predicate logic and its design is essentially based around the principles of the algebraic approach. The formal semantics of COLD-1 are defined in terms of the COLD-K kernel language, although it does include a number of extensions, such as: user definable infix and postfix operators, patterns for procedure definitions, patterns for components, and a variety of naming mechanisms. According to Feijs [FEI94a], COLD-K comprises six sub-languages at the syntactic level: a name language, an expression language, an assertion language, a definition language, a scheme language, and a design language. Whereas, at the semantic level, COLD-K comprises three logical systems for giving meaning to the kernel language: the logic MPL_ω, the algebra CA, and the $\lambda\pi$ calculus.

Specification under COLD-1 is based on the conventional view that a software system comprises a collection of states; one initial state and a number of state transformers - i.e. procedures. In COLD-1, state components are precisely defined structures and come in three different forms: sorts (i.e. collections of objects), functions (i.e. partial mappings from domain sorts to range sorts), and predicates (i.e. relations on domain sorts). Hence, states are modelled as many-sorted

[9] H.B.M. Jonkers, *An introduction to COLD-K*. In M. Wirsing and J.A. Bergstra (Eds.), *Algebraic Methods: Theory, Tools and Applications*, Springer-Verlag LNCS 394, pp. 139-205, Springer-Verlag, 1989.

algebras [FEI94b, page 40]. A software system, then, is modelled in COLD-1 by definitions of sorts and operations, as well as various axioms and theorems which characterise defined sorts and operations. There are three possible kinds of operation: *functions* are mappings from one data type to another, *predicates* describe properties of data types, whilst *procedures* are operations which are affected by and affect states. The inclusion of constructs for describing system behaviour via equational axioms and pre- and post-conditions enables COLD-1 to express either algebraic or model-oriented style specifications. The notation provides mechanisms for importing, hiding and parameterising specification modules which are optionally parameterisable and may contain: definitions, axioms, theorems, lists of other modules which may be imported, and lists of hidden sorts and operations. The design principles underlying COLD-1's modular structuring mechanisms were as follows:

1. To enhance the comprehensibility of the specification.
2. To make reasoning about the specification easier.
3. To improve the adaptability of the specification.
4. To make reuse of the specification possible.

Adapted from Feijs, Jonkers and Middleburg [FEI94b, pages 325-326].

The "CLASS" construct is a central concept to a COLD-1 specification. A class can be viewed as an abstract machine with a number of associated machine states. These states are many-sorted algebras with predicates and (partial) functions, with all of the states of a function having the same signature. A class definition contains an interface part (containing an export signature, an import clause and preconditions), and a body (containing imports, operation definitions and a list of modification rights). A "scheme" is the result of modularising and parameterising a set of operations. Finally, schemes that are combined to form a functional component constitute a subsystem [LAV94]. The following simple COLD-1 example shows one of the preliminary, abstract specifications produced by Feijs et al. [FEI94b] for describing the components of a vending machine system. Here, the COLD-1 reserved word "FREE" denotes a forward declaration item - i.e. definition of its precise structure will be postponed until a later stage, when the specification is refined.

```
COMPONENT VENDING2 SPECIFICATION
IMPORT
    VENDING1
    VALUE
CLASS
    SORT ValueTable FREE
    SORT PriceTable FREE
    SORT ProdTable FREE

    FUNC validate : ValueTable # Coin → Value FREE
    FUNC price : PriceTable # Selection → Value FREE
    FUNC yield : ProdTable # Selection → Product FREE

    FUNC value_table : → ValueTable FREE
    FUNC price_table : → PriceTable FREE
    FUNC product_table : → ProductTable FREE

    FUNC vend : Coin # Selection → Product
    IN c,s
    DEF ( price(price_table,s) <= validate(value_table,c) ?
        ; yield(product_table,s)
        )
```

```
END
```

Feijs, Jonkers and Middleburg [FEI94b, page 12].

The following example employs a wider range of COLD-1's constructs, and illustrates how various properties of Boolean sorts and operations can be characterised through the use of axioms and theorems.

```
COMPONENT BOOL SPECIFICATION
EXPORT
    SORT   Bool                          % names of sort and
    FUNC   false :              -> Bool, % functions
           true :               -> Bool,
           not  : Bool          -> Bool,
           and  : Bool # Bool   -> Bool
CLASS
    SORT Bool
    FUNC false   : -> Bool               % the constructors of Bool
    FUNC true    : -> Bool

    DECL b,c : Bool

    PRED is_gen : Bool          % inductive definition of
    IND      is_gen(false);     % is-generated predicate
             is_gen(true)

    AXIOM false! AND true!;     % axiom characterising
             NOT false = true;  % "!", "=" and "Bool"
             is_gen(b)

    FUNC not : Bool -> Bool     % inductive definitions of
    IND      not(false) = true; % negation and conjunction
             not(true) = false

    FUNC and : Bool # Bool -> Bool
    IND      and(false,false) = false;
             and(false,true) = false;
             and(true,false) = false;
             and(true,true) = true

    THEOREM not(b)! AND and(b,c)! % theorem about definedness
END
```

Feijs, Jonkers and Middleburg [FEI94b, page 330].

COLD-1 also supports algorithmic definitions (in an imperative or functional style), which effectively allow designers to construct explicit implementations. Accordingly, the language incorporates a number of imperative programming constructs in order to ease the program transformation process. State components can thus comprise entirely specification oriented descriptions or both specification and implementation parts. The modelling of explicit implementations normally requires designers to consider algorithms with regard to program efficiency and performance. In the following example, for instance, the specification part describes what the "MACHINE" component does, in terms of three sorts and four procedures. In contrast, the implementation part describes how the machine achieves this in terms of technical implementation modules [FEI94, pages 89-91].

```
COMPONENT MACHINE

SPECIFICATION
EXPORT
   PROC   insert  : Coin      - > ,
          check   :           - > ,
          select  : Selection - > ,
          produce :           - > Product
IMPORT
   VENDING
END

IMPLEMENTATION
IMPORT
   COIN,
   VALUE,
   SELECTION,
   PRODUCT,
   HASHMAP'[Selection,Product,max2,hash2]
   :
END
```

Adapted from Feijs, Jonkers and Middleburg [FEI94b, page 90].

Wherever implementation parts are specified, designers must ensure that an "implementation relation" exists between the specification and implementation parts. Firstly, the signature exported by the specification part must be a subset of the signature exported by the implementation part. Secondly, all facts and properties described in the specification part must be logical consequences of the definitions given in the implementation part. The process of establishing this implementation relation nearly always gives rise to proof obligations.

3.8 CSP

Communicating Sequential Processes (CSP) was developed by Hoare [HOA85] and his fellow members of the Programming Research Group at Oxford University during the early 1980s. It originated mainly in response to some of the difficulties being experienced during the design of parallelism in software relating to the likes of: mutual exclusion, semaphores, interference, interrupts and asynchronous behaviour. It comprises both a specification notation and a deductive proof system, whereby the latter is defined in the form of inference rules.

The CSP notation is based on elementary algebra, set theory and the predicate calculus. It is a process algebra which models a system as a collection of independent processes. Each process is defined in terms of the possible interactions that it may have with other processes and the environment - invariably, the environment itself is modelled as just another process. The interactions between processes are described in terms of instantaneous atomic synchronisations, or "events". Any two processes may intercommunicate providing both are ready and have been suspended in order for a synchronised transfer of data to take place - this is termed a "rendezvous". Designers can impose constraints upon processes by restricting the set of names denoting the events in which they may engage, the order in which events occur and the conditions under which they may execute. As the concerns of CSP specifications are constrained purely to the behaviour of processes and their intercommunication, all internal system state information is effectively abstracted away.

CSP enforces a strictly bottom-up approach to design; only after one has identified all of the various processes and sub-processes that comprise a target system

can one begin to postulate their various interrelations and behaviour. The notation is one of the most popular process algebras in existence and has been used not only as a tool for specifying concurrent systems, but its language mechanisms have been greatly influential in the progression of empirical software and hardware knowledge. In particular, CSP's underlying "rendezvous" mechanism has influenced the design of processor hardware (an Inmos transputer, for example) and the definition of programming languages (such as Ada and Occam).

The notion of "prefixing" is central to the CSP notation. If there exists a process P and a is any event from P's alphabet then $a \rightarrow P$ denotes a process which first engages in event a then behaves like the process P. Choice is expressed using the "|" operator. For example, $(x \rightarrow P \mid y \rightarrow Q)$ first engages in event x or event y, as determined by the environment. Aside from choice based on input values, various alternative forms of choice are supported such as: choice by channel, prioritised choice and non-deterministic choice. More than one process may be joined to form a single, composite process via sequencing or parallel composition. In addition, processes may be recursively defined so as to permit the expression of iteration, which may or may not terminate [OU90]. The following example provides a small flavour of CSP's extensive notation. It describes the behaviour of a vending machine system which dispenses a coffee drink or chocolate bar depending upon whether the user inserts a small or large coin, respectively. It also permits the user to insert two small coins (rather than one large coin) in exchange for a chocolate bar, and returns the correct change in the event of a user inserting more than an item's value.

$$
\begin{array}{lll}
VM = (\mathit{large} & \rightarrow (\mathit{choc} & \rightarrow VM \\
 & \quad \mid \mathit{coffee} & \rightarrow \mathit{small}\ VM)) \\
\quad \mid \mathit{small} & \rightarrow (\mathit{coffee} & \rightarrow VM \\
 & \quad \mid \mathit{small} & \rightarrow (\mathit{choc} \rightarrow VM))
\end{array}
$$

Adapted from Hoare [HOA85, page 30].

The foundations for the CSP notation itself lie in a number of different areas including: the predicate calculus, set theory, standard mathematical tools (such as functions), Hoare's own notational symbols (such as those for processes, traces and special events) and elementary algebra. The notation provides constructs for describing all aspects of process behaviour including: sequencing, alternation, parallelism and recursion. However, one potential drawback to readability is the large number of parentheses that can appear within even the simplest of CSP expressions. Designers might therefore avoid cluttering expressions by selectively delimiting the number of parentheses used and rely upon their readers resolving any possible ambiguities with recourse to CSP's algebraic laws.

3.9 Clear

The Clear notation was developed by Burstall and Goguen during the late 1970s. Although it is intended primarily as a tool for program specification, it also serves to represent knowledge (i.e. "theories") in algebraic, machine readable form. Before discussing the notation itself, it is first necessary to introduce some terminology. A Clear specification basically consists of two parts; a set of syntactic declarations (called a "signature"), and a set of axiomatic declarations (the equational part). A "signature" comprises a set of "sort" name declarations (one for each sort of data involved, such as *nat* or *bool*) and a set of operational symbol declarations (such as a unary successor operation on *nat* called "*succ*"). The axiomatic part comprises a series of equations expressing the relations between certain pairs of terms introduced in the signature [BUR81]. Hence, a Clear "theory" comprises a signature and a set of equations closed under inference by reflexivity, transitivity

and symmetry of equality and by substitution [BUR77]. The following example illustrates an algebraic theory of natural numbers and Boolean values.

```
sorts       nat, bool
operations  zero   : → nat
            succ   : nat → nat
            iszero : nat → bool
            true   : → bool
            false  : → bool
            not    : bool → bool
            or     : bool, bool → bool
variables   m,n    : nat
equations   iszero(zero) = true
            iszero(succ(n)) = false
            not(true) = false
            not(false) = true
```

Burstall and Goguen [BUR77].

The Clear notation contains theory building operations which enable designers to construct "theory expressions" denoting complex interrelated theories.[10] In Clear, it is possible to express parameterised specifications, called "procedures", which take theories as arguments and return theories as values. Such procedures impose requirements on their arguments in the form of axioms which must be true before their application becomes meaningful; these requirements are themselves theories [BUR81]. Program specifications in Clear, then, comprise constant and procedure declarations, followed by a sequence of theories constructed using Clear's theory-building operations and the previously declared constants and procedures. The following example shows a more comprehensive theory procedure for the sort "Stack". Any type of element can be put onto a stack - this is indicated by its acceptance of a trivial theory, called "Triv," as an input parameter.[11]

```
proc Stack (Value : Triv) =
  induce enrich Value + Bool by
    sorts       stack
    opns        nilstack  : → stack
                push      : value, stack → stack
                empty     : stack → bool
                pop       : stack → stack
                top       : stack → value
    erroropns   underflow: → stack
                undef : → value
    eqns        empty(nilstack) = true
                empty(push(v,s)) = false
                pop(push(v,s)) = s
                top(push(v,s)) = v
    erroreqns   pop(empty) = underflow
                top(empty) = undef
                pop(underflow) = underflow
  enden
```

Burstall and Goguen [BUR77].

[10] For details of the "combine", "enrich", "induce" and "derive" constructors, the reader is directed to Burstall and Goguen [BUR77].

[11] The doubly underlined terms are themselves theory procedures defined elsewhere, each satisfying the obvious. Clear theories often become hierarchically interdependent in this way.

Burstall and Goguen speculate that perhaps the "main intellectual task of programming is elaborating the theories which describe all the concepts used in the actual program" [BUR77]. Clear is a language for communicating precise problem specifications to people, aimed mainly at programmers, who are more likely to immediately recognise some correspondence between a Clear specification and a program's structure. In view of the way in which Clear specifications focus on sorts and the operations on those sorts, this makes Clear well suited to the specification of abstract data types in particular. Naturally, complex software systems require the development of complex theories, which in turn contain numerous subtheories - some of these specify particular data structures and operations, while others assert axioms which may be satisfied by any of a variety of structures. This is the philosophy upon which the Clear notation is founded.

It is generally believed that certain types of formalism are more appropriate for the development of certain kinds of application owing to the nature of their underlying logical systems and formal semantics. However, the Clear language has been defined in such a way as to allow different underlying logical systems to be used, so long as they satisfy certain conditions regarding the relationship between theories and models. Burstall and Goguen refer to such logical systems as "institutions" [BUR81, SAN91]. Some common institutions include equational logic (for describing general purpose system behaviour), first order logic (for describing concurrent systems), temporal logic (for describing error-handling behaviour) and continuous equational logic (for describing non-terminating behaviour).

3.10 Extended ML

Extended ML provides a framework for the development of executable Standard ML programs from property oriented, algebraic specifications. It is commonly referred to as a "wide-spectrum" language because its notation can be used to formally describe a program throughout all stages of its development, including those intermediate stages where it contains both specification and program parts. Extended ML extends the Standard ML programming language by allowing the inclusion of axioms in module interfaces and their substitution for code in module bodies. These axioms are used to put constraints on the permitted behaviour of components and can be used as a basis for proving program correctness. The Extended ML framework was initially developed at the University of Edinburgh during the mid-1980s. The main aims of its language designers were as follows:

1. *Formality.* To use a formal calculus to prove that the outcome of a program development process is correct with respect to a requirements specification.
2. *Methodology.* To define a number of development steps which may be applied to specifications of a certain form, together with a list of conditions which must be established in order to guarantee correctness.
3. *Modularity.* Large programs should be built in modular fashion from small and relatively independent program units. The Extended ML framework should support this approach and, in turn, promote module reusability.
4. *Machine Support.* Computer-aided tools should support the various software development activities and to help minimise the possibility of human error.

Adapted from Sannella [SAN91].

Typically, development under the Extended ML framework proceeds firstly by producing incomplete program modules. Here, the term "incomplete" is used in the sense that some parts of the program are specified only by means of axioms, rather than Standard ML executable programming constructs. Thus, at various

stages of development, an Extended ML program might contain both non-executable (abstract) and executable elements. Each refinement step gives rise to one or more proof obligations, which must then be proven in order to establish the validity of that refinement. Program development under the Extended ML framework concludes when all of the specification axioms have been translated into the corresponding Standard ML code. The following specification is an example of an sorting routine for a sequence of elements, and is expressed in the Extended ML notation.

```
functor Sort(X : PO) : sig include SORT
                sharing Elements = X
            end =
    struct
        structure Elements : PO = X
        datatype sequence =
            nil
          | cons of Elements.elem * sequence
        fun append(nil,s) = s
          | append(cons(a,s1),s2) = cons(a,append(s1,s2))
        fun member(a:Elements.elem,s:sequence) = ? : bool
        axiom member(a,nil) = false
        axiom member(a,cons(a,s)) = true
        axiom a<>b => member(a,cons(b,s)) = member(a,s)
        fun insert(a:Elements.elem,s:sequence) = ? : sequence
        axiom member(a,insert(a,s))
        axiom insert(a,s) = append(s1,cons(a,s2))
            => append(s1,s2) = s
            andalso (member(a1,s1) => Elements.le(a1,a))
            andalso (member(a2,s2) => Elements.le(a,a2))
        fun sort nil = nil
          | sort(cons(a,s)) = insert(a,sort s)
    end
```

Sannella [SAN91].

Although first-order equational logic is used in the example specification shown, the formal underpinnings of Extended ML are theoretically independent of the logic (or "institution") used. However, for practical reasons, it is wise to select a logic that has the Standard ML core language as a subset. This has the advantage of enabling one to easily recognise that program development has finished; when all of the specification axioms have been translated to this executable language subset [SAN91]. According to Sannella, Extended ML's semantics view executable code as a special form of axiomatic description. So, Extended ML can theoretically be used to develop programs in target programming languages other than Standard ML. For example, the Extended ML framework could support the development of modular Prolog programs by switching to untyped first-order predicate logic and by viewing Horn clauses as the executable subset of this logic.

3.11 Gypsy

Gypsy provides a notation, called "Program Description Language" (PDL), and an environment for conducting deductive proofs, called "Gypsy Verification Environment" (GVE). Gypsy was first developed at the University of Texas in 1974, and is aimed in particular at supporting the development of reliable concurrent communicating systems. The Gypsy notation is designed to enable the description of a program throughout all stages of its development - i.e. from initial specification through to final implementation, verification and any subsequent evolution.

It, thus, incorporates both specification language facilities (based around the predicate calculus) and a high-level programming language (which is based on the syntax of Pascal). According to the language's designers, Ambler et al. [AMB77], the incorporation of specification and programming facilities into a single language is beneficial to the program verification process in three ways. Firstly, it means that formal proofs can be constructed before program execution occurs. Secondly, specifications can be validated by execution at run-time. Finally, Gypsy's trace facilities support post-execution program analysis. According to Ambler et al., the following represent some of the specific goals that Gypsy's designers had hoped to achieve from the language.

1. *Complete verifiability.* All features of Gypsy must be fully verifiable either by formal proof or validation at run-time.
2. *Incremental development.* The language must support incremental program development and simplify verification by encouraging the use of small, logically distinct units that can be verified independently.
3. *Systems programming.* Gypsy must support the development of systems software (including process concurrency, synchronisation and real-time dependencies).
4. *Imperfect execution environments.* Gypsy must allow for the fact that programs are rarely executed in perfect execution environments and, hence, the language must also provide facilities for error detection, isolation and recovery.
5. *Specification capability.* The language must provide extensive features for the purpose of specification - formal proof, run-time validation and monitoring must form a well integrated whole.

Adapted from Ambler et al. [AMB77].

Although PDL was originally derived from the Pascal language, it underwent several extensive modifications in order to provide a sufficiently flexible paradigm for incremental program development. In particular, facilities were added for expressing concurrency, communication, synchronisation, timing constraints, external events, error recovery and monitoring. The specification component of the language includes the predicate calculus, recursive functions, and enables specifications to be written as Floyd-Hoare style program annotations, algebraic style axioms and abstract state machine descriptions.

A Gypsy program consists of a series of independently verifiable units, which may be functions, procedures or programs. Each unit contains a header section describing the unit's pre- and post-conditions. Access rights to any unit are specified in an "access list". Gypsy processes communicate through finite length message buffers and are invoked using a "cobegin" statement. The concept of real-time is provided by "clock" variables which are a special kind of internal variable that are constantly being updated. In addition, there are a large number of predefined "conditions" which correspond to hardware errors and semantic language errors.

Gypsy's specification language component permits the description of functional properties of software systems. These properties are stated in terms of valid states that are to be maintained on the data objects of the program at various points in the program's execution. All Gypsy specifications are expressed as Boolean-valued statements - these may be verified either by proof, by run-time validation, or they may simply be assumed. Specifications need not be evaluated at run-time and, hence, they need not be parsed and may freely contain various constructs that would not otherwise be permitted in executable programs - such as logically quantified statements. The following example shows a Gypsy program specification for the channels of communication between a network and its external environment.

```
program Network(var upa:PortArray) = pending;

type PortArray = array(UserId) of Port;

type UserId = integer(1..NUsers);
const NUsers:integer = pending;

type Port = record(Get,Put:Line);
type Line = buffer(CSize) of Message;
const Csize:integer = pending;

type Message = pending;
```

Ambler, Good, Browne, Burger, Cohen, Hoch, and Wells [AMB77].

The following example shows a Gypsy procedure specification for an operation which writes a sequence of bytes to a data file.

```
procedure WRITE_TO_SEQUENCE (in_file : file;
                             offset  : natural;
                             data    : byte_seq) : byte_seq =
begin
    exit (result =
        if (offset le size (in_file))
            then in_file (1..offset-1) @ data
            else in_file @ n_zeros ([offset - size (in_file)] + 1)
                @ data
        fi;
end; (write_to_sequence)
```

Young [YOU89].

The kind of abstraction provided by Gypsy is called "procedural" because related data items are combined into single modules. However, Young [YOU89] is critical of this approach and argues that, once such items have been associated with a particular structure, this effectively leads to a biased implementation because it is extremely difficult to re-associate data items later on. Also, if the corresponding programming language implementation is structurally different from a specification's grouping of related data items, then the procedural nature of a Gypsy specification can, in fact, obscure rather than clarify the mapping between specification and program code. Other notations overcome this problem in various ways. For example, the Z notation provides "schema" abstraction, where individual data items may be included in different schemas and no correspondence to particular implementation structurings are implied. Young is also opposed to the idea that all data structures accessed by a Gypsy module must either be locally defined or else passed in as parameters because this means that module headers tend to become easily cluttered and extremely verbose. Although, this does have the advantage that Gypsy modules can always be understood in isolation, without need to reference their calling environment. In comparison, to gain a full understanding of a Z schema, one might need to expand all of the schemata in the hierarchy beneath it.

The overall procedure for development under the Gypsy framework consists of writing a specification in the PDL notation then expanding each of its units into explicit programs. The correctness of each program unit can then be proven (with respect to its specification unit) with the aid of GVE's integrated tool set. GVE comprises a collection of tools for creating, specifying, maintaining and verifying Gypsy programs. These tools include a parser, verification condition generator, interactive proof checker, algebraic simplifier and language compiler. Gypsy appears

well suited towards the specification and verification of concurrent, communicating systems, such as network applications. However, one distinguishing feature of Gypsy's specification of real-time systems is that it relies upon the histories of message buffers in order to describe process behaviour, as opposed to other types of formalism which use conventional state transition methods. Gypsy concentrates purely on the external or input/output behavior of processes and includes language constructs for referring to the history of messages read and written by the processes in each buffer.

3.12 IOTA

The IOTA system provides a modular software development environment, under which program modules may be specified, verified, compiled and maintained independently. It was developed at Kyoto University's Research Institute for Mathematical Science, Japan. IOTA's integrated development system contains five major subsystems, each of which are coordinated via the system's central component, called its "modulebase".

1. *Developer.* This analyses, checks then translates the given source text into an appropriate representation for storage in the system's modulebase.
2. *Debugger.* An abstraction-oriented dynamic debugging facility.
3. *Verifier.* An interactive verification system specially designed for modular system development.
4. *Prover.* An interactive proof system for a certain many-sorted first-order logic.
5. *Executor.* This generates executable code from the inner representations of modules, which is then loaded and executed.

Adapted from Yuasa and Nakajima [YUA85].

There are three types of IOTA module: "type" modules contain abstract data type definitions, "procedure" modules contain definitions of functions which operate on data types, whilst "sype" modules specify the parameters of parameterised type and procedure modules. All IOTA module types include an interface part, containing declarations of the operations introduced by the module together with their domains and ranges, and a specification part, consisting of axioms in a many-sorted first-order (predicate) logic. Type and procedure modules also contain an additional "realization" part, which contains an implementation of the module to satisfy the axioms given in its specification. The following example shows Yuasa and Nakajima's English requirements description and formal specification for their "Keyword in Context" application.

> "Given a list of titles (e.g. book titles) and a set of words called nonsignificant words, the function *kwic* generates an alphabetically sorted list of all rotations of the titles that begin with significant words."

INTERFACE PROCEDURE *kw*
  FN *kwic: (list(title),set(string)) → list(title)*
END INTERFACE

SPECIFICATION PROCEDURE *kw*
  VAR *titles:list(title); nonsigs:set(string); t,t1:title;*
  AXIOM 1: *is_sorted(kwic(titles,nonsigs))*
  2: *element_of(t,kwic(titles,nonsigs))*
  $\Leftrightarrow t = \backslash = nil$
  $\wedge \neg in(first(t),nonsigs)$

```
                    ∧ ∃t l.(element_of(t1,titles) ∧ is_rotation_of(t,t1))
END SPECIFICATION

REALIZATION PROCEDURE kw
   FN kwic(titles:list(title); nonsigs:set(string))
   RETURN (sigrots:list(title))
      VAR rots:list(title);
      sigrots := nil;
      WHILE titles ≠ nil DO
         rots := get_all_rotations(head(titles));
         WHILE rots ≠ nil DO
            IF ¬in(first(head(rots)),nonsigs)
               THEN sigrots := insert(sigrots,head(rots))
            END IF;
            rots := tail(rots);
         END WHILE;
         titles := tail(titles);
      END WHILE
   END FN
END REALIZATION
```

Adapted from Yuasa and Nakajima [YUA85].

According to the IOTA system development philosophy, a specification should be regarded only as "partial" because it describes only part of the properties required by a program realization - i.e. specifications abstract away various implementation details. Thus, the notion of completeness in IOTA is regarded as a undecidable property for specifications given by first order axioms. Owing to the fact that a specification may be partial, there may exist many different versions of a specification even for the most trivial of implementations. Hence, Yuasa and Nakajima claim that this flexibility promotes the use of specifications as well defined descriptions of individual modules, as opposed to complete (but vague) descriptions of entire systems. However, the specification is still regarded as a mechanism to aid program development rather than as a means of documentation. A specification provides a module's declarative description, whilst an implementation provides a module's procedural description. Both are complementary and, according to Yuasa and Nakajima, the parallel development, modification, and verification of each makes it an easier task to guarantee correctness.

Having recognised that modification of a module on large scale development projects might affect other developers who, directly or indirectly, refer to that module, IOTA's designers included facilities to aid developers in maintaining consistent interrelations between module interfaces. The language was designed in such a way as to limit the effects of modifications as locally as possible so that details of updates to the affected developers would be transmitted promptly. Modular programming requires proofs to be done on a relatively large number of axioms. Theoretically, the verification of a single module could involve the axioms specified in all of the modules that the unit under verification refers to. Thus, Yuasa and Nakajima admit that module verification in IOTA can often result in a lengthy formula amounting to pages to be proved about only a few axioms, without knowing for certain whether this really is a theorem. One of the primary aims of IOTA's designers was to discover how an integrated formal specification and verification environment would affect traditional programming activities. According to Yuasa and Nakajima [YUA85], IOTA was developed for "experimental and educational" purposes uses only, and was never intended to be practically applied to large scale, complex software development projects. If, however, IOTA were to be amended for use on industrial applications, then its program verification techniques would need

to be simplified considerably.

3.13 Larch

Larch was developed in the main by Guttag and Horning [GUT85a, GUT85b] at MIT laboratories, during the early 1980s. Unlike most other formalisms, the term "Larch" does not refer to one specific formalism, but to a whole range of formal specification notations, each one being based on existing programming languages. Every Larch specification comprises two parts; one is based on a specific programming language, the other is general to all languages. The former is called a Larch "interface" language, the latter is the Larch "shared" language.

A Larch interface language is used to specify program components, with particular attention being paid to the interaction between components and their environment. Thus, only a program's externally observable behaviour is specified, as opposed to its internal workings. An interface language allows a designer to write assertions about program states, which can be translated into predicate calculus formulae. It also provides facilities for describing general programming issues, such as: side effects, exceptions and iteration.

The Larch shared language is used to formally define the terms introduced in interface specifications. According to Guttag et al. [GUT85a], it generates "theories" that are independent of any particular programming language and which are free from implementation detail. Shared language specifications are expressed in algebraic form, with equations defining the relations between operators. Central to the Larch shared language is the notion of a "trait", which is used to introduce operators and describe their formal properties - where sets of operators serve to describe abstract data types. The following trait illustrates how a Larch shared language can be used to specify a class of tables that may store values in indexed positions. The first part of the specification (following the "introduces" clause) declares a set of operators (corresponding to program function identifiers), each with its own signature - i.e. the sorts of its domain and range. The equations presented in the second part effectively constrain the operators introduced in the first part by relating terms containing operators.

Tablespec: trait
  introduces
    new: $\rightarrow$ Table
    add: Table, Index, Val $\rightarrow$ Table
    # $\in$ #: Index, Table $\rightarrow$ Bool
    eval: Table, Index $\rightarrow$ Val
    isEmpty: Table $\rightarrow$ Bool
    size: Table $\rightarrow$ Card
  constrains new, add, $\in$, eval, isEmpty, size so that
    for all [*ind*, ind_1: Index, *val*: Val, *t*: Table]
      eval(add(*t*, *ind*, *val*),ind_1) = if *ind* = ind_1 then *val* else eval(*t*, ind_1)
      ind $\in$ new = false
      ind $\in$ add(*t*, ind_1, *val*) = (*ind* = ind_1) | (*ind* $\in$ *t*)
      size(new) = 0
      size(add(*t*,*ind*,*val*)) = if *ind* $\in$ *t* then size(*t*) else size(*t*) + 1
    isEmpty*(*t*) = (size(*t*) = 0)

Guttag, Horning and Wing [GUT85a, page 6].

One of the main roles of a shared language specification is to define the theories that will eventually be implemented in executable form. Larch's trait mechanism provides a means for achieving this because a theory may informally be associated with each trait declaration.

"A theory is a set of well-formed formulas of typed first-order predicate calculus with equations as atomic formulas. The theory, call it *Th* associated with a trait written in the Larch shared language is defined by:

- Axioms: Each equation, universally quantified by the variable declarations of the containing constrains clause, is in *Th*.
- Inequation: $\neq$ (true = false) is in *Th*. All other equations in *Th* are derivable from this one and the meaning of =.
- First-order predicate calculus with equality: *Th* contains the axioms of conventional typed first-order predicate calculus with equality and is closed under its rules of inference".

Guttag, Horning and Wing [GUT85a].

Every Larch interface language is designed around a particular programming language so that there is a clear correspondence between concepts in the two languages - the meanings of an interface language's reserved words, for instance, are derived from their corresponding meanings in the programming language. The following is an example of data abstraction in a Larch/Pascal interface specification:

```
type Bag exports bagInit, bagAdd, bagRemove, bagChoose
  based on sort MSet from Multiset with [integer for E]
  procedure bagInit(var b: Bag)
    modifies at most [b]
    ensures b_post = {}
  procedure bagAdd (var b: Bag; e: integer)
    requires numElements(insert(b, e)) ≤ 100
    modifies at most [b]
    ensures b_post = insert(b,e)
  procedure bagRemove (var b: Bag; e: integer)
    modifies at most [b]
    ensures b_post = delete(b,e)
  function bagChoose (b: Bag; var e: integer) : boolean
    modifies at most [e]
    ensures if ¬IsEmpty(b)
      then bagChoose & count(b, e_post) > 0
      else ¬bagChoose & modifies nothing
end Bag
```

Guttag, Horning and Wing [GUT85a, page 15].

The "Bag" interface specification describes which routines must be implemented, indicating the pre-condition and post-condition for each. This constitutes a form of contract between client and developer. The client must ensure that the conditions described in the "requires" clause are met at each point of call and may assume the truth of the "ensures" clause on return from the routine, whereby only those variables listed in the "modifies at most" may be changed. The developers are entitled to assume the truth of the "requires" clause on entry to a routine and must themselves make sure that the conditions specified in the "ensures" clause are true on return, whilst making sure that only those variables that are explicitly listed in the "modifies at most" clause are altered.

A Larch "two-tiered" specification differs from conventional model based specifications in several significant ways. Firstly, Larch's shared language is used to specify theories rather than models. Secondly, interface languages are built around the predicate calculus rather than operational notations. This, according to Guttag et al. [GUT85a], means that Larch based specifications are less prone to implementation bias than many other formal notations. Owing to the fact that there

exist substantial differences between programming languages (say, with regard to exception signalling, parameter passing, and storage allocation), their corresponding Larch interface languages also differ widely. Larch interface languages have already been designed for a variety of programming languages including: Ada, C, C++, CLU, ML and Smalltalk. Larch is also supported by a tool set for generating deductive proofs which comprises: a shared language syntax checker, an interface specification checker and a theorem prover.

3.14 LOTOS

During the late 1980s, the International Organisation for Standardisation (ISO) developed a range of methods for the description of data communication protocols and services, called "Open Systems Interconnection" (OSI). It was recognised that, in order for OSI to become recognised as a genuine standard it must be supported with an appropriate formal description technique in which its protocols and services could be specified and verified. An international committee set out to produce this formal technique which eventually resulted in the creation of the language LOTOS. Since becoming registered as an international standard in 1989.[12] However, LOTOS has proved useful not only in the specification of OSI protocols and services, but also in the development of a wide range of distributed systems. The following principles were central to the design of LOTOS:

1. *To provide complementary formalisms for "data" and "control".* No single existing formalism was considered general enough to express conveniently both the control and data components of a specification. Accordingly, LOTOS' designers sought to incorporate the appropriate parts from two formalisms: ACT ONE for the data part, and CCS/CSP for the control part.
2. *To produce a precisely and formally defined notation.* LOTOS includes a formally defined syntax with static semantics (defined by an attributed grammar) and dynamic semantics (described operationally in terms of inference rules).
3. *To obtain all the deductive proof capabilities of a conventional process algebra.* The operational semantics of LOTOS were defined in such a way as to enable designers to prove a large and varied range of algebraic equivalence properties based on various types of equivalence relations.
4. *To implement the principle of interleaving concurrency.* Events are seen as atomic actions and the parallel execution of two events a and b is defined as a situation of choice, where a can occur before b, or *vice versa*. Any LOTOS behaviour expression can be rewritten as an expression consisting of a choice between behaviour expressions, each prefixed by a single action.
5. *To achieve specification executability.* Since its semantics are defined operationally, it is possible for LOTOS specifications to be machine interpreted or semi-automatically translated to program code. Executable specifications can be viewed as *fast prototype* models of a system under specification.
6. *To achieve specification modularity and module reusability.* LOTOS favours stepwise decomposition of processes and their parameterisation promotes their reusability.

   Adapted from Logrippo, Faci and Haj-Hussein [LOG90b].

LOTOS' static semantics are defined by an attributed grammar, whilst its dynamic semantics are based on algebraic concepts. LOTOS' data type component is based on the algebraic data type formalism ACT ONE, which permits the definition of data structures and value expressions. Its control component is based on

[12] International Standards Organisation, *LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* ISO Information Processing Systems, Open Systems Interconnection, ISO International Standard 8807, 1988.

a mixture of Milner's CCS [MIL89] and Hoare's CSP [HOA85], and permits the description of process behaviour and communication channels. LOTOS' dynamic semantics for the control component are expressed in operational terms by inference rules, and its operators enable one to prove a wide range of algebraic equivalence properties. Thus, the language is to some extent "executable" (by virtue of its operational semantics) and amenable to proof techniques (by virtue of its algebraic properties) [LOG90a].

An action in LOTOS contains three components: a gate, a list of events and an optional predicate. If two processes name the same gate, their event lists match and any supplementary predicates are satisfied, then they can synchronise on the named gate. Events can offer (!) or accept (?) values and predicates can be used to describe the precise conditions under which values can be offered or accepted. Actions that can be executed independently by a process and which also require synchronisation with the environment are represented by the internal action *i*. These actions are offered at synchronisation points called "gates". When an action is executed, the behaviour expression of the process is transformed into another behaviour expression. It is the inference rules of LOTOS' dynamic semantics that determine which actions are offered and executed by a process and how behaviour expressions are transformed by effect of the execution of actions [LOG90b].

Where a process declares itself ready to synchronise with other processes and establish one or more values, this is termed an "action offer". For example, "$g?x$: *integer*" states that the process is ready to synchronise on any integer value with other processes, on gate "*g*". Action offers can be combined through the use of several operators: "[ ]" (choice), "|[A]|" (parallel execution with synchronisation via gates in set A), "||" (parallel execution with synchronisation on all gates), "|||" (parallel execution with interleave; no synchronisation), "*hide*" (abstract hiding of gates), ">>" (sequential composition of processes)[13], and "[>" (disable, modelling non-deterministic interruption). There exist two behaviour expressions which denote the state of a process: "*stop*" denotes deadlock, and "*exit*" denotes a process's successful termination. Behaviour expressions can be preceded by guards ("[ ]"), which must be true in order for the former to be allowed to execute.

```
specification Max[in1, in2, in3, out] : noexit

type integer is
    sorts int
    opns
        zero : → int
        succ : int → int
        largest : int, int → int
    eqns forall X, Y : int ofsort int
        largest(zero, X) = X;
        largest(X, zero) = X;
        largest(succ(X), succ(Y)) = succ(largest(X, Y));
endtype

behaviour
    hide mid in
    (
        Max2[in1, in2, mid]
        |[mid]|
        Max2[mid, in3, out]
    )
```

[13] In $P1 >> P2$, the *enable* operator ">>" expresses sequential composition between the two processes, although its semantics differ slightly from those of the action prefix operator ";" depending upon whether an "*exit*" event occurs in process $P1$.

```
where
   process Max2[val1, val2, max] : noexit :=
      (val1?X : int; exit(X; any int)
      |||
      val2?Y : int; exit(any int, Y)
      )
      >> accept V : int, W : int in
         max!largest(V, W); stop
   endproc
endspec
```

Adapted from Logrippo, Melanchurch and DuWors [LOG90a].

According to Logrippo et al., there are four main styles in which a LOTOS specification can be presented: monolithic, state-oriented, resource-oriented and constraint-oriented. However, the different styles can obviously be mixed depending upon the type of application being specified and the document's audience.

1. *Monolithic.* The main operator used is the "[ ]" choice operator, while parallel composition operators are not used and the specification is written as a "tree" of alternatives.
2. *State-oriented.* Explicit state variables are used.
3. *Resource-oriented.* Specification modules are selected in such a way as to identify potential modules for implementation.
4. *Constraint-oriented.* Processes identify families of constraints and the parallel composition of processes specifies the simultaneous satisfaction of all constraints.

Adapted from Logrippo, Melanchurch and DuWors [LOG90a].

Based on its application to large-scale, complex software projects, Logrippo et al. drew several conclusions with regard to LOTOS' practical application in industry:

- Establishing the correspondence between LOTOS events and real-life events can be problematic.
- LOTOS promotes precise reasoning about a specification.
- LOTOS helps identify incompleteness and inconsistency in requirements.
- LOTOS forces the meticulous resolution of ambiguities in a specification.
- Simulated execution leads to a greater appreciation of the total interdependencies between components.
- It is probably easier to introduce translation errors when implementing from a LOTOS specification.
- Stylistic concerns are just as important as for traditional programming languages; it is possible in LOTOS, as in any other language, to write confusing complex specifications or programs.

Adapted from Logrippo, Melanchurch and DuWors [LOG90a].

Owing to its basis on CSP and CCS, LOTOS represents a considerably expressive notation. Both the data type and the control part of LOTOS are each sufficiently powerful to describe complex systems. It is largely the decision of the specifier how to balance the two, but the way in which this trade-off is solved can greatly influence the ease of writing a specification and its readability. It encourages highly modularised, structured specifications and supports both a top-down as well as a

bottom-up approach to design. Furthermore, since its standardisation, it has been applied extensively in industry for the specification of communication protocols, and is continually undergoing further research in academia. A number of utilities are currently available to support the development of LOTOS specifications. These include: a language interpreter, symbolic expander, theorem prover, abstract data type persistency checker, trace checker, and automated code generator.

3.15 OBJ

OBJ derives its name from one of the central conceptual entities of the notation, that is, the "object". It was initially conceived during the late 1970s as a means for testing algebraic specifications of abstract data types, although it has become widely used as a general purpose specification language for producing executable prototypes and for defining programming language semantics. OBJ's underlying grammatical semantics are based on equational logic, whilst its operational semantics are based upon "rewrite rules".[14] The OBJ language is aimed at achieving four specific design goals: modular and parameterisable specifications, strong typing and subsorts, implementation facilities, user interaction and language flexibility. Furthermore, according to Goguen and Tardo [GOG79], OBJ is designed at overcoming several specific problems that have traditionally become associated with algebraic design languages: typographical errors, imprecise notations, failure to handle special cases (such as boundary cases), specifications that give rise to different results if applied in different ways, specifications that fail to terminate when they should, and failure to handle error conditions.

OBJ encourages reusability by supporting parameterisable modules. These modules are effectively generic because they can be used in different ways simply by changing the types of their parameter values. A high degree of interaction is required during program entry where users must select from a series of structured menus; this is in contrast to non-interactive forms of program development, such as keyword entry and subsequent machine parsing. The Command Interpreter Generator (CIG) is OBJ's built-in application generator which facilitates the generation of prototype systems and allows various forms of experimentation with OBJ's syntax and semantics. The language also enables users to create their own abstract descriptions of program properties, using any desired syntax, and subsequently to use these abstractions as if they were built-in. For example, OBJ's "mixfix" syntax permits prefix, postfix, infix, outfix and distributed fix operations, in which keywords and arguments may be placed in any order. If desired, users can provide efficient implementations of these abstractions in executable Lisp code.

An OBJ specification consists of a series of "modules" and "views". A module can either be an "object" (with optional parameters, but always containing executable code) or a "theory" (which contains non-executable logical assertions). Views are used to relate theories to modules. This provides for a framework under which both executable and non-executable statements can be modularised and closely linked. "Requirement theories" are used to specify the syntax and axioms for a module's interface - only those objects which satisfy the criteria defined by

[14] Equations in some high-level, executable languages (such as OBJ and Prolog) can be viewed as "rewrite rules". That is, an equation $\forall X.t = t'$ can be viewed as a rewrite rule of the form $t \Rightarrow t'$ (or $t' \Rightarrow t$). This means that any substitution instance of t in an expression can be replaced by the corresponding substitution instance of t'. When no further pattern matches or substitutions are possible, the resulting expression is said to be in "normal form". This process of rewriting can, under certain conditions, be used to compute the value of an expression. For example, the "conditions" that an equation must satisfy in order to qualify as a valid rewrite rule might include the requisite that it is Church-Rosser compliant and that the rewriting process will always terminate in a unique result [SAN93].

these axioms are admissible. The following example shows a requirement theory for a module, "SORTING," which sorts lists over an ordered set X.

th POSET *is*
  protecting BOOL .
  sort Elt .
  op _ < _ : Elt Elt → Bool .
  vars E E' E" : Elt .
  eq : E < E = false .
  ceq : E < E" = true
    if (E < E' and E' < E") .
endth

Futatsugi, Goguen, Jouannaud, and Meseguer [FUT85].

Objects are typically used to introduce new sorts of data and new operations which may manipulate that data. An object comprises three parts: a header (containing the object's name, its parameters, its interface requirements and a list of its imported modules), a signature (containing new sort declarations, subsort relationships and operations), and a body (containing code which is built from equations and constraints on sorts). The following example shows a parameterised sorting object - the type of value passed to this routine determines the type of element that the routine may sort.

obj SORTING /ELT :: POSET/ *is*
  extending LIST/ELT/ .
  op sorted : List → Bool .
  op sort : List → .
  vars E E' : Elt .
  var L L' L" : List .
  eq : sorted(nil) = true .
  eq : sorted(E) = true .
  eq : sorted(E E' L) =
    (E == E' or E < E') and sorted(E' L) .
  ceq : sort(L E L' E' L") =
    sort(L E' L' E L") *if* E' < E .
  ceq : sort(L) = L *if* sorted(L) .
jbo

Futatsugi, Goguen, Jouannaud, and Meseguer [FUT85].

In order to instantiate a parameterised object, one must provide actual objects satisfying each of its requirement theories. Views provide these actual objects by binding the required sorts and operations to those specified. For example, the following view can be used to provide the necessary bindings to the previously defined SORT object in order to sort integers into descending order.

view INT-DESC *of* INT *as* POSET *is*
  sort Elt *to* Int .
  vars X Y : Elt .
  op : X < Y *to* : Y < X .
endview

Futatsugi, Goguen and Jouannaud [FUT85].

OBJ, like most other algebraic formalisms, allows designers to specify only a limited amount of detail with regard to data types - i.e. names may be given, but very few structural details. Similarly, no precise algorithms may be specified for

operations; only the various equational interrelations with other operations. This effectively leads to minimal implementation bias. One definite advantage of employing the OBJ formalism is that its specifications can be machine checked for syntactic correctness, compiled, then subsequently animated. Thus, certain sources of error may be automatically detected after only a short time of having initially created a specification. So, to some extent, OBJ may be regarded as a non-procedural, applicative programming language but with little concern for system performance. Users may employ the Lisp programming language to append additional functions or objects to OBJ and then annotate these with OBJ constructs and equations. This is the conventional method used to provide OBJ with its built-in objects for describing commonly used data types (Booleans and arrays, for example) and number systems (such as natural or real numbers). This method effectively combines the efficiency of a compiled programming language with a mathematical description of what the function or object does - the latter can be useful for theorem proving.

The OBJ environment provides extensive testing facilities for: verifying the correctness of objects in isolation, testing the interconnections between objects, and checking the use of error conditions. It is supported by additional tools such as the 2OBJ theorem prover[15] which provides facilities for checking that equations are Church-Rosser compliant and will terminate following rewriting, and for checking that actual parameter objects satisfy their corresponding requirement theories. Although most previous applications of OBJ have been limited mainly to academic use (such as the teaching of software design, rapid prototyping and theorem proving techniques), it has been successfully applied in industry for verifying the correctness of various hardware components.

The OBJ language has undergone numerous modifications since its first release. It was originally conceived in the form of OBJ-0 by Tardo and Goguen in 1977 and was later implemented in the form of OBJ-T [GOG79]. This early work was influenced by Goguen's work on the Clear notation which appeared at around the same time. In fact, OBJ is often regarded as an implemented, executable subset of Clear. Improvements were subsequently made to OBJ's rule-rewriting capabilities and its interactive dialogue, and these were incorporated into Plaisted's implementation of OBJ-1.[16] Further work by Goguen and his colleagues at Stanford University during the early 1980s culminated in the OBJ-2 language [FUT85]. OBJ-3, the current version of the language, builds on the facilities provided by its predecessors, and provides additional support for error handling, multiple inheritance, overloading, and user-definable execution strategies. As it stands, the OBJ-3 notation incorporates the results of over twenty years' research in algebraic semantics.

3.16 PVS

PVS (Prototype Verification System) was developed at SRI International during the late 1980s. It integrates a highly expressive specification notation (based on higher-order logic) with an interactive theorem prover and various supporting tools (such as a parser, type checker and pretty printer). SRI entitled these development tools the "Prototype Verification System" because they were built primarily in order to experiment with state-of-the-art verification technology for the company's main verification system, EHDM (Enhanced Hierarchical Development Method[17]). The general aims of the language's designers were to create a system which supports

[15] A. Stevens, *2OBJ.* Programming Research Group Technical Report Number TR-2-94, Oxford University, 1994.

[16] J.A. Goguen, J. Meseguer, and D. Plaisted, *Programming with parameterised abstract objects in OBJ.* In D. Ferrari, M. Bolognani, and J. Goguen (Eds.), *Theory and Practice of Software Technology*, pp. 163-193, North-Holland, 1983.

[17] P.M. Melliar-Smith and J. Rushby, *The Enhanced HDM system for specification and verification.* In ACM Software Engineering Notes, 10 (4), August 1985.

clear and abstract specifications, and provides an overall framework for constructing readable and correct proofs of complex theorems.

A PVS specification consists of a series of theories, each comprising a sequence of declarations which provide names for types, constants, variables, axioms and formulas. A number of primitive theories are built into the system; these are collectively referred to as the PVS "prelude". By allowing for theories to be parameterised, this effectively provides support for specification modularity and reuse. Owre et al. [OWR92] state that the design of the PVS notation was influenced by a variety of formal notations including: Affirm, Gypsy, OBJ, RAISE, VDM and Z. Yet, a strongly typed, higher-order logic underlies the notation. Crow et al. [CRO95] claim that higher-order logic encourages the writing of "compact and perspicuous" specifications. A conscious decision of PVS's designers had been to mechanise the verification process as far as possible and, at the same time, complement the highly expressive nature of its underlying logic. Indeed, the notation's foundations in higher-order logic make for a highly expressive, abstract, specification language. The following PVS specification shows a description of the requirements for an electronic phone book system.

```
phone : THEORY

  BEGIN

  N: TYPE                          % names
  P: TYPE                          % phone numbers
  B: TYPE = [N -> setof[P]]        % phone books
  nm, x: VAR N
  pn: VAR P
  bk: VAR B

  emptybook(nm): setof[P] = emptyset[P]
  FindPhone(bk, nm): setof[P] = bk(nm)
  AddPhone(bk, nm, pn): B = bk WITH [(nm) := add(pn, bk(nm))]
  DelPhone(bk, nm): B = bk WITH [(nm) := emptyset[P]]
  DelPhoneNum(bk, nm, pn): B = bk WITH [(nm) := remove(pn, bk(nm))]
  FindAdd: CONJECTURE member(pn, FindPhone(AddPhone(bk, nm, pn), nm))
  DelAdd: CONJECTURE DelPhoneNum(AddPhone(bk, nm, pn), nm, pn) =
                                    DelPhoneNum(bk, nm, pn)

END phone
```

Adapted from Crow, Owre, Rushby, Shankar, and Srivas [CRO95].

PVS contains a sophisticated type system; its predefined data types include: number systems, enumerations, records, tuples, arrays, functions, sets, sequences, lists and trees. By using such type constructs, the notation's designers were forced to compromise the degree to which its specifications can be mechanically checked; PVS's rich and flexible type system effectively makes mechanical type checking of its specifications algorithmically undecidable. Its type checker does, however, compensate by generating appropriate proof obligations for the PVS theorem prover. PVS includes a number of features designed to promote program correctness. These include strong type checking, consistency preserving mechanisms and a theorem prover. The theorem prover, a central PVS component, supports top-down proof exploration and construction. It implements a set of primitive inference rules, a means for applying these to proof strategies, a mechanism for performing proofs, and a facility for checking that all secondary proof obligations have been discharged. The activities involved in proving a specification correct conform to a general life cycle of the form outlined below:

1. *Exploratory phase.* The initial concern is for checking a specification and its putative theorems, and testing the key ideas in the proof.
2. *Development phase.* Next, details of the proof are considered and an overall proof strategy begins to evolve.
3. *Presentation phase.* The proof is polished for presentation and others' scrutinisation.
4. *Generalisation phase.* The finished proof is analysed and its assumptions are weakened or generalised, making it easier to carry out similar verifications in future.
5. *Maintenance.* Here, a verification may require slight changes following a change to assumptions or requirements.

Adapted from Owre et al. [OWR92].

Despite initially being developed for the purpose of prototyping new verification technologies, PVS has been adopted by companies world-wide and used successfully to verify various types of application. These range from complex algorithms for fault tolerant systems to microcode for driving various hardware devices.

3.17 RAISE

RAISE is an acronym for Rigorous Approach to Industrial Software Engineering. It was initially used as the name for an industrial research and development project carried out as part of the ESPRIT programme during the late 1980s. However, the name has become used to refer to a specification language, a design method and a set of proof tools developed by the RAISE Language Group. RAISE is aimed specifically at facilitating the practical application of formal methods in industry. The RAISE development method provides a general framework and a set of guidelines for a range of software engineering activities, such as capturing requirements and project management. It is presented in a sufficiently flexible manner to be adaptable to a wide range of software development projects and allows a user to choose the level of formality appropriate to the particular circumstances. It is based on the "Stepwise Refinement" development methodology, whereby software is constructed through a series of steps and each step represents a refinement of its predecessor. The style of refinement supported by RAISE is known as "theory extension". According to this principle, in simple terms, a theory $T2$ is an extension of another $T1$ providing $T2$ contains more entities and, possibly, more properties than $T1$.

Development of the Raise Specification Language (RSL) was motivated by model-oriented notations (VDM, in particular, for its specification structuring mechanisms), algebraic notations (such as Larch for its abstract, axiomatic descriptions of system properties), and process algebras (such as CCS and CSP for expressing concurrency). The foundations for the notation therefore lie in a number of diverse areas including the predicate calculus, set theory and equational algebra. So, despite its syntactic and semantic similarities with a range of other formalisms, RSL is not confined to the specification of one particular type of system - such as purely sequential or purely concurrent systems. RSL claims to support a wide variety of specification types, including the following:

- *Model-oriented* specifications, as used in Z and VDM. Under this style, any data available to a specification is 'modelled' - for example a stack may be modelled as a sequence of values. Operations are described by how they affect the data model.
- *Algebraic* specifications, as used in Larch and OBJ. This is a more abstract method of specification than the model-oriented approach. Under this approach, data is described using axioms which specify its various properties. For example, a stack may be defined by saying that $pop(push(x, s)) = s$ for all stacks s and elements x. This allows for a wide range of possible models to satisfy the axioms. Within

the RAISE method, an algebraic specification is generally developed into a model-oriented one as more concrete data structures are added.

- *Applicative* specifications, similar to the style used in applicative programming languages such as Lisp, Miranda and Haskell. Using this style, there is no concept of a global execution history, machine state or variable definitions. The result of the application of any function to its parameters is merely a function of the parameters to the function. So, every application of a function to the same parameters will have the same result.
- *Imperative* specifications, similar to the style used in imperative programming languages such as C, Modula-2 and Pascal. With this style, a system's execution history and global state are prominent. Variables can be defined and changed and, thus, the order in which procedures and functions are applied has an important effect on the results.
- *Concurrent* specifications, as used in CSP. Concurrent processes can be defined which communicate with each other via synchronous channels.
- *Implicit* definitions, using pre-conditions and post-conditions.
- *Explicit* definitions, describing internal functionality.

Adapted from Hellinger [HEL95].

RSL specifications are organised into a series of modules, each of which is basically a named collection of declarations or "class expressions". A class expression represents a class of models, where each model associates an entity (such as a value, type, variable, channel or module) with each identifier defined within the class expression. Hence, RSL class expressions share a number of semantic similarities to conventional theory presentation constructs (i.e. signatures and axioms) in algebraic specification notations. Through modularisation, RSL's language designers aim to promote the comprehensibility and reusability of specifications. RSL supports two kinds of module: objects and schemes. An object is a named model chosen from a class of models and is represented by a class expression. Whereas, a scheme is simply a named class expression [RAI92, pages 204-212]. By naming class expressions, this allows designers to manipulate them more easily through the use of RSL's in-built operations - such as extending, renaming or hiding. The class expression construct and the RSL notation itself is illustrated in the following RSL specification of a parameterised ordered list.

```
scheme
  PARAM_ORDERED_LIST(T : PARTIAL_ORDER) =
    class
      variable
        list : T.Elem*
      value
        empty : Unit → write list Unit,
        is_empty : Unit → read list Bool,
        add : T.Elem → write list Unit,
        head : Unit ~→ read list T.Elem,
        tail : Unit ~→ write list Unit
      axiom forall e : T.Elem •
        empty() ≡ list := <>,
        is_empty() ≡ list = <>,
        add(e) ≡ list := <e> ^ list,
        head() ≡ hd list
        pre ~is_empty(),
        tail() ≡ list := tl list
        pre ~is_empty()
      value
```

```
            is_ordered : Unit → read list Bool
        axiom
            is_ordered() ≡
                (∀ idx1,idx2 : Nat •
                    {idx1,idx2} ⊆ inds list ∧ idx1 < idx2 ⇒
                        T.leq(list(idx1),list(idx2)))
object
    TEXT :
        class
            type
                Elem = Text
            value
                eq : Text x Text → Bool,
                leq : Text x Text → Bool
            axiom forall t1,t2 : Text •
                eq(t1,t2) ≡ t1 = t2,
                leq(t1,t2) ≡ (∃ t : Text • t1 ^ t = t2)
            end
object TEXT_LIST : PARAM_ORDERED_LIST(TEXT)
```

The RAISE Language Group [RAI92, pages 224-225].

Both the RAISE method and its notation are supported by an integrated tool set, which provides facilities for: editing, verifying designs or specifications, generating code and system documentation. Although RAISE provides automated tool support for the verification process, a designer may elect to perform the verification manually, using rigorous argument, instead. LaCoS (Large scale Correct Systems), the successor to the ESPRIT RAISE project, has concentrated on applying RAISE to industry based projects. Its aim is to demonstrate that RAISE is a commercially viable technique and is well suited to the production of large scale, reliably correct software systems. RAISE has already been successfully employed in the development of such diverse applications as: a process monitoring system, a train management system and an image processing system. Its only potential weakness, according to Hellinger, is that it is generally not suited to the specification of systems involving non-functional requirements, such as temporal or spatial constraints.

3.18 The Refinement Calculus

Morgan's Refinement Calculus [MOR90] combines the predicate calculus with the works of Back, Dijkstra, Hoare, Floyd and Morris. Its foundations lie in Dijkstra's weakest pre-condition method and language of "Guarded Commands", however, there are a number of notable extensions. The calculus was developed during the late 1980s at Oxford University and, despite having received considerable attention from software theorists in academia, Morgan's calculus has generated limited interest from engineers in industry and has yet to be fully applied to a large scale software development project. Morgan regards a specification as "a contract between a programmer and his client" and immediately discards any potentially confusing distinctions between abstract specifications, subspecifications, and executable programs by viewing every item in the development hierarchy as a "program" [MOR90, page 1]. The following illustrates the form of a top level program specification expressed in the calculus:

$$w : \left[pre, post \right]$$

Where *pre* is the program's pre-condition, *post* is the post-condition, and w is the frame - i.e. a list of variables that may change during the course of the operation.

The general aim of the refinement calculus is to take a top level program specification S and, by applying Morgan's rules for refinement at a number of intermediate stages, gradually refine it to executable code C in some target programming language. According to Morgan, each refinement introduces either "a little more executability or a little more efficiency". This results in a systematic program derivation sequence, illustrated as follows:

$$S \sqsubseteq M_0 \sqsubseteq M_1 \sqsubseteq \cdots \sqsubseteq M_{n-1} \sqsubseteq M_n \sqsubseteq C$$

For instance, the following example shows the initial, top level specification for a factorial program and the resulting executable code obtained after applying Morgan's rules of refinement.[18]

$$
\begin{array}{ll}
 & f : [0 \leq n, f = n!] \\
\sqsubseteq & \ldots \\
\sqsubseteq & \textbf{procedure } \mathit{Fact} \mathrel{\widehat{=}} \\
 & \quad \textbf{if } n = 0 \rightarrow f := 1 \\
 & \quad [] \ n > 0 \rightarrow \\
 & \quad\quad \mathit{Fact}[\textbf{value } n : \mathbb{N} \setminus n \ominus 1]; \\
 & \quad\quad f := f \times n \\
 & \quad \textbf{fi}
\end{array}
$$

Adapted from Morgan [MOR90, pages 122-123].

Owing to its reliance upon the standard predicate calculus and typed set theory, Morgan's calculus presents a suitable paradigm for refinement in state based notations such as Z and VDM. Morgan argues that the predicate calculus is useful to computer scientists, but only where long formal proofs are not required. Hence, when devising the Refinement Calculus, Morgan set out to: minimise the number of proof obligations required at each refinement step, use sophisticated methods for logical argument which have already been proven by others (such as proof by contradiction), and leave it to the discretion of the designer as to which parts of a specification to prove formally and which ones to leave as "intuitively obvious". Thus, he does not advocate any particular method of formal logical reasoning with the predicate calculus - such as natural deductive, axiomatic or tableaux. Rather, Morgan's use of the predicate calculus in reasoning about specifications is evident in his laws for refinement which express equalities between formulae and may be used to simplify complex formulae [MOR90, pages 15-16].

3.19 VDM

Having originally been developed at IBM's Vienna Research Laboratories in the early 1970s, the Vienna Development Method (VDM) is one of the longest established formal methods. VDM provides a specification notation (called "VDM-SL") and a set of procedures which cater for the whole software development process, including a proof theory which can be used to verify properties of VDM specifications. By allowing designers to use abstraction at the highest level of a system's description, VDM effectively encourages top-down development. A VDM specification consists of two major parts. The first part contains a set of abstract data type and variable definitions which represent the internal data within a system model. This is termed the "system state". The second part comprises definitions of the operations and functions which manipulate the variables in order to achieve the

[18] For details of the intermediate stages, see Morgan [MOR90, page 122].

requirements of the customer. These are specified using predicate calculus expressions and concepts from set theory. The following example shows the requirements for a marriage bureau's database system:

State :: UNMARRIED :*Person-set*
MARRIED :*Person-set*
Person = /* some suitable representation */

inv-State $\widehat{=}$ unmarried $\cap$ married = {}

REGISTER (P: *Person*)
ext UNMARRIED : **wr** *Person-set*
MARRIED : **rd** *Person-set*
pre p $\notin$ unmarried $\wedge$ p $\notin$ married
post unmarried' = unmarried $\cup$ {p}

MARRY (M: *Person*, W: *Person*)
ext UNMARRIED : **wr** *Person-set*
MARRIED : **wr** *Person-set*
pre m $\in$ unmarried $\wedge$ w $\in$ unmarried
post **let** couple = {m, w} **in**
married' = married $\cup$ couple $\wedge$
unmarried' = unmarried - couple

INIT()
ext UNMARRIED : **wr** *Person-set*
MARRIED : **wr** *Person-set*
post unmarried' = {} $\wedge$ married' = {}

Adapted from Cohen [COH86, pages 56-58].

The system state is represented using two abstract data structures, "UNMARRIED" and "MARRIED". As can be seen from the definition of "*Person*", designers are free to delay the precise definition of a data type's structure until a later stage, perhaps when the customer's exact requirements have been determined. The remainder of the system state specification describes the externally visible operations which may be invoked by the marriage bureau system's users. Each operational specification contains the following elements:

1. The operation's name and any input or output parameters.
2. An "**ext**" clause which indicates those parts of the state that the operation needs to access and the read/write access permissions for each state component accessed.
3. The pre-condition, indicating the conditions for which the operation is designed to have an effect.
4. The post-condition, showing how the values of the state's variables are affected by the operation and how the values of the output parameters are to be generated.

The general method for developing programs from a VDM specification is to proceed through various levels of design, progressively adding more implementation detail at each stage. As abstract data structures are refined into forms more closely resembling programming language concepts (such as sets into trees, or trees into arrays) new versions of the specification's operations must be formulated in terms of the new data structures to correspond with those at the more abstract level. Eventually, data structures will become specified at a sufficiently low level for program code to be easily derivable. At each point of refinement, "retrieve functions" are formulated in order to verify that every value of the abstract type can

be represented in the converted structure - this notion is called "adequacy". These functions also make it possible to verify that the refined operations correctly model the effects of the abstract operations that they are intended to mimic. Following the final refinement step, standard program proving techniques can be employed to check that the code produced correctly fulfils the least abstract specification.

VDM can be classified as a state based technique because it represents systems as abstract models which pass through various states. A model contains data objects representing inputs, outputs, the system's internal state, operations and functions that manipulate the data. Unlike Z, with which it shares numerous similarities, VDM provides little support for modularity - this is provided in Z by the schema facility. Furthermore, VDM specifications are not easily accessible to novice users and, according to Hull and O'Donaghue, tend to lack concision [HUL93]. Since its initial development, VDM has been important in influencing the development of many other formalisms and has been used extensively in industry and academia, especially in the area of programming language definition. It has also undergone numerous developments which have transformed it into a general purpose software development method, making it sufficiently flexible to be applied to a variety of applications, such as databases and operating systems. At the time of writing, VDM-SL is undergoing further enhancements which aim to provide features for describing concurrency. The notation is currently being reviewed for standardisation by ISO and there also plans to provide VDM specifications with an execution environment in which executable subsets of the language may be animated.

3.20 Z

Origins of the Z notation can be traced back to Abriel's preparatory work during the late 1970s [ABR80], however, this was later developed at Oxford University by Hayes [HAY87] and Spivey [SPI92]. Additional work culminated in a base standard for the Z notation [BRI92], which also provides a deductive proof system for reasoning about Z specifications. The foundations for the Z notation lie in the propositional and predicate calculi, as well as typed set theory, although there are a number of notable extensions in the form of a well defined "mathematical toolkit". This toolkit provides a variety of formally defined constructs including: functions, relations, schemas, sequences and bags. These allow program data types to be described at an abstract level. Z also includes various operators for manipulating these constructs.

The format of a Z specification generally comprises a several formal sections, some of which are optional, with complementary interspersions of informal, explanatory text. Firstly, a number of basic types are declared, which represent those sets whose existence is assumed - these serve to abstract away from the precise structures of a program's data types. Secondly, a series of axiomatic descriptions are given in order to declare those constants that are global to the whole specification. Next, a number of user-defined sets are given. Following this, the system's state space is specified, which details the invariant properties of a system, along with a description of the initial system state. Finally, a series of operations are defined, which describe the valid state transitions that may occur in the system.

The Z "schema" structuring facility provides a means for creating modular and reusable specifications. Each of the logically distinct sections of mathematical text can be placed in a separate schema and, thus, clearly distinguished from a document's natural language background. A schema contains two optional parts: a declaration part and a predicate part. A new schema can be constructed by including the names of previously defined schemas in the former's declaration section. Z specifications therefore tend to exhibit a form of hierarchical schema structure. Aside from inclusion, additional schema operators are available for extension, restriction and composition. There are three possible kinds of Z schema: state space schemas,

initial state schemas and operational schemas. State space schemas are used to specify the invariant properties of a system. The following example delineates the state space of of a vending machine system, showing the various interrelations between each of the system's main conceptual units.

VMStockDatabase

$$Itemcodes : CODE \rightarrowtail ITEM$$
$$Stocklevels : ITEM \twoheadrightarrow LEVEL$$
$$Itemprices : ITEM \rightarrow PRICE$$

$$ran\ Itemcodes = dom\ Stocklevels = dom\ Itemprices$$

An initial state schema specifies the conditions that must hold before a system begins executing. The following example states that all item codes, stock levels and item prices should not be defined prior to the commencement of the vending machine's operation.

Initialise

$$StockDatabase'$$
$$UserCoins'$$

$$ran\ Itemcodes' = dom\ Stocklevels' = dom\ Itemprices' = \varnothing$$
$$Usercredit' = <>$$

The following example shows a Z operational schema for the item dispensing operation of the vending machine system. Inputs to the operation are signified with a "?" suffix, whilst output variables are indicated by a "!" suffix. References to predefined schemas are indicated by the inclusion symbol "Δ" in the schema's declaration section.

DispenseItem

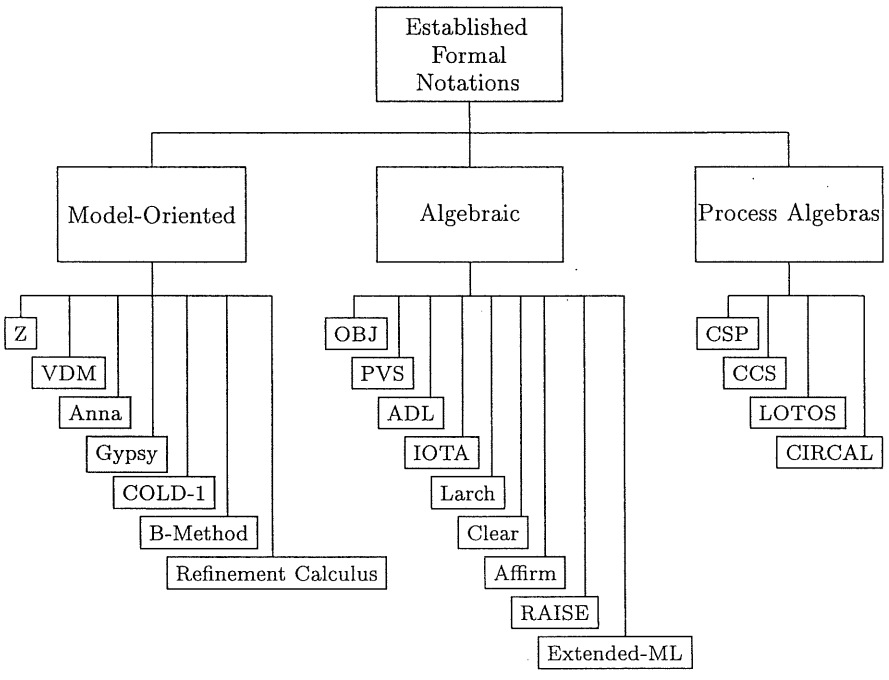
$$\Delta StockDatabase$$
$$\Delta UserCoins$$
$$Inputcode? : CODE$$
$$Stockitem, Outputitem! : ITEM$$

$$(Inputcode? \mapsto Stockitem \in Itemcodes \land$$
$$Usercredit \geq Itemprices\ Stockitem \land$$
$$Stocklevels\ Stockitem \geq 1)$$
$$Stocklevels' = Stocklevels \oplus \{Stockitem \mapsto (Stocklevels\ Stockitem - 1)\}$$
$$Usercredit' = Usercredit - Itemprices\ Stockitem$$
$$Outputitem! = Stockitem$$

A pre-condition may be specified for each operational schema in order to define the conditions under which each one becomes applicable - a pre-condition of "true", for example, states that an operation is applicable under all circumstances. Although the Z and VDM notations appear to differ widely, their underlying specification methods are relatively similar. Both provide for implicit specification of operations using pre- and post-conditions, and state invariants. Furthermore, both methods' observable inputs and outputs are represented using standard mathematical constructs such as sets, functions, and sequences. Since a Z specification builds an explicit model representation of a desired system and, owing to its reliance upon local and global state changes, state variant and invariant properties, Z is classifiable under the model-oriented approach to specification.

One of the principle aims of the Z notation's designers was to provide its users with a language for creating precise and easily understandable program specifications and yet, interestingly, Z provides very little graphical or diagrammatic support - in comparison with, say, Petri net based notations. In fact, the schema's two-dimensional box notation is the only element of graphical notation supported. Perhaps in compensation for Z's relatively obscure symbology, the language's standard definition allows users to include informal natural language descriptions within specifications. Overall, Diller [DIL94, page 68] believes that a Z specification should generally comprise more English prose than formal mathematics and that, accompanying each section of Z, there should be a paragraph of English narrative which elaborates on its formal counterpart and relates it to the real world - that is, it does not simply translate the formal Z to an equivalent form in English. One further aim of its designers was to enable designers to build system models at two different levels of abstraction. Firstly, "representational abstraction" can be expressed using Z's high-level mathematical constructs (such as sets, relations, functions and sequences) to model program data types, without giving any consideration to the way in which these will eventually be constructed in an implementation. Secondly, it is possible to achieve "procedural abstraction" by specifying operations in terms of their input and output behaviour only, without giving any consideration to the efficiency of the algorithms underlying those operations.

4 A Summary of the Review



The diagram gives an overview of the various notations reviewed and shows their respective categorisations under the classes of model-oriented, algebraic and process algebra techniques. It should be noted that two particular classifications are based on subjective assessment. Firstly, Gypsy is classified here under the model-oriented approach because its PDL notation allows properties of programs to be described in terms of valid states that are to be maintained at various stages during a program's execution. However, it is also potentially capable of describing properties

of programs via abstract, axiomatic statements. Gypsy therefore has strong algebraic roots and is equally classifiable under the algebraic division. Secondly, whilst RAISE's RSL notation contains elements of all three classes of specification technique, it is the author's belief that RSL's theoretical foundations and grammatical constructs coincide mainly with those underlying the class of algebraic languages.

Whilst the present review might appear exhaustive with regard to the three main classes of specification notation, its scope is relatively narrow in the context of formal methods overall; a large number of other methods exist and new ones are continually evolving as a result of ongoing academic research. Specifically, some of the more popular notations excluded from the review include the following:

1. Functional programming languages (such as the Lambda Calculus[19] and Miranda[20]).
2. Petri net or graph based notations (such as SARA[21] and Galileo[22]).
3. Rewrite rule based or logic programming languages (such as Prolog[23] and Gödel[24]).
4. Experimental notations based on modal, temporal or interval constraint logics.

To justify the exclusion of such notations from the present survey, in the light of the Human Cognition and Formal Methods project's specific research aims, three forms of justification are given. Firstly, with a few exceptions, many of these techniques are either still in their infancy or else have not gained the same degree of popular academic or industrial acceptance exhibited by those techniques falling under the aforementioned trichotomy of approaches. Secondly, the commercial viability of many of these formalisms has yet to be proven and, hence, their current degree of usage in industry is extremely limited - this is especially true with regard to functional languages and the new breed of languages based on modal logic. Finally, the kind of symbology exhibited by a number of these languages is not generally representative of the full range of notations - take, for example, the diagrammatic graph style exhibited by Petri net based notations.

5 Comments on the Review

One of the main intellectual problems facing the designers of a formal specification language concerns how to define the notation's semantics so that they are sufficiently powerful to define the semantics of software systems in specifications, or possibly even the semantics of further specification languages. This is known as the "metacircularity problem" and is described by Monahan and Shaw [MON91]. If the underlying semantics of a notation's denotational constructs are either inconsistent or ill-defined, then it is highly likely that the ideal degree of metacircularity will not be achieved; because one could never guarantee that every well-formed specification has a corresponding logical interpretation. The crux of the problem is concerned with delimiting the class of entities that may populate the denotation of a specification. According to Monahan and Shaw, this "presents a similar mathematical challenge to that which arises in giving a mathematical definition of the foundations of mathematics itself" [MON91].

[19] M.C. Henson, *Elements of Functional Languages.* Blackwell Scientific Publications, 1987.

[20] I. Holyer, *Functional Programming With Miranda.* Pitman, 1991.

[21] I.M. Campos and G. Estrin, *Concurrent software system design supported by SARA at the age of one.* In *Proceedings of the Third International Conference on Software Engineering. Atlanta, Georgia*, 1978.

[22] F. Vidondo, I. Lopez, and J.J. Girod, *Galileo system design method.* In *Electrical Communications, 55* (4), 1980.

[23] I. Bratko, *Prolog Programming for Artificial intelligence.* Second Edition, International Computer Science Series, Addison-Wesley, 1990.

[24] P.M. Hill and J.W. Lloyd, *The Gödel Programming Language.* MIT Press, 1994.

> "It is no accident that the basic data types used in model-based specifications of software are useful in describing computer-oriented systems. These data types were carefully and deliberately chosen by the specification language designer as appropriate abstractions of familiar computer-oriented concepts which commonly arise in practice."
>
> Monahan and Shaw [MON91].

Language designers aim to represent the denotational constructs of specification languages with simple mathematical constructs for a number of reasons. Firstly, the "basic data types" to which Monahan and Shaw refer tend to include basic mathematical concepts such as sets, Cartesian products and relations, which are taught normally during childhood education. Thus, it is assumed that a reader's increased familiarity with denotational constructs will be more likely to result in their interpretation of a formal specification coinciding with that of its logically correct meaning. At first sight, the symbology of some formal notations can appear quite daunting to some readers. However, upon closer inspection, readers are generally able to recognise that formal specifications are built entirely from simple, well understood primitives. This is perhaps one of the main obstacles that the readers of formal specifications face, but it is often this recognition which helps readers to overcome their initial apprehensions and encourages them to piece together each primitive's meaning in order to derive a specification's overall meaning. Secondly, elementary mathematical data types are used because their syntax tends to be concise and their underlying semantics are rigorously defined. This promotes the production of succinct specifications which are open to precise, unique interpretations. Thirdly, the use of simple mathematical data types enables the computer-oriented concepts under specification to be represented at a sufficiently high level of abstraction. This is often highly desirable because a specification should ultimately leave a design team free to consider a range of possible implementations, rather than constraining them unduly.

During the course of the review, it was observed that many formal notations are based around the same or similar types of logic system to those underlying other notations. The RAISE RSL notation, for example, exhibits striking similarities to VDM and Larch; not only do these notations share the same underlying calculi, but they share many of the same grammatical constructs. In fact, in a majority of the cases observed, it was found that formal notations are rarely developed from first principles. Typically, the design theories, logical calculi, semantical rules and syntactical components comprising a new notation are either truncated or extended versions of those belonging to other languages. This trait is perhaps most evident in the case of "primitive" formal languages used as a basis for the definition of more "enhanced" versions. For instance, consider Object-Z[25] and Real-Time CSP.[26] These represent enhanced versions of the more primitive Z [SPI92] and CSP [HOA85] languages, respectively, and inherit the full expressive power of their primitive counterparts. Despite some notable differences in the languages' syntactical constructs, the formal logics underlying each are actually quite similar; Object-Z and Z, for instance, are both based upon the predicate calculus and set theory, and both adopt a model-oriented approach to design specification. The Z notation and the B-Method's AMN notation also possess similar logical foundations - both being based on the predicate calculus and set theory - and yet their symbology differs dramatically. Compare, for instance, the two languages' specification structuring mechanisms (Z's schema versus AMN's abstract state machine) and the differences

[25] R. Duke, P. King, G. Rose and G. Smith, *The Object-Z Specification Language: Version 1.* Department of Computer Science, University of Queensland, Technical Report 91.1, April 1991.

[26] J. Davies and S. Schneider, *Real-time CSP: Processes and Properties.* Department of Computer Science, University of Reading, Reading. RG6 2AH. November, 1993.

in their grammatical constructs (Z's complex mathematical symbols versus AMN's ASCII compliant character set).

One further conclusion drawn from the review is that there exists a strong correlation between the design theories underlying formal notations and the logical systems on which they are based. For instance, most model-oriented notations tend to have firm foundations in the predicate calculus, often with set-theoretic and mathematical extensions, whilst process algebras tend to be based on purpose-built, strictly behaviour oriented logical systems - such as Milner's synchronous calculus - also with set-theoretic and mathematical extensions. This raises the question of whether specific types of logical system are more amenable to the description of certain types of program behaviour. Intuitively, the predicate calculus would appear better suited to the description of state-based program behaviour than, say, the synchronous calculus, which would be more suited to expressing concurrent or communicating behaviour. This might explain why, in practice, certain formal notations are used exclusively for specifying certain types of target application. For example, it is well known that CCS and CSP have been used successfully in modelling protocols of communication across networked computer systems, whilst other formalisms, such as Z and VDM, have proved more useful in modelling the kind of behaviour exhibited by the likes of information processing systems.

Acknowledgements

The author would like to thank the following reviewers for their many helpful comments regarding this report: Ben Potter, Martin Loomes and Jane Simpson. The author would also like to thank all of those authors cited during the course of this report whose work formed the basis of the review. The Human Cognition and Formal Methods research project is supported by Grant No. J00429434043 from the Economic and Social Research Council.

References

[ABR80] J.R. Abriel, *The specification language Z: basic library*, Programming Research Group, Oxford University Computing Laboratory, 11 Keble Road, Oxford. OX1 3QD, 1980.

[AMB77] A.L. Ambler, D.I. Good, J.C. Browne, W.F. Burger, R.M. Cohen, C.G. Hoch, and R.E. Wells, *Gypsy: A language for specification and implementation of verifiable programs*, ACM SIGPLAN Notices, *12* (3), March 1977.

[BCO95a] B-Core (UK) Limited, *The B-Technologies.* B-Core (UK) Ltd, Magdalen Centre, The Oxford Science Park, Oxford. OX4 4GA. March, 1995.

[BCO95b] B-Core (UK) Limited, *The B-Technologies: A system for computer aided programming.* B-Core (UK) Ltd, Magdalen Centre, The Oxford Science Park, Oxford. OX4 4GA. May, 1995.

[BRI92] S.M. Brien et al., *Z Base Standard. Version 1.0*, Programming Research Group, Oxford University Computing Laboratory, 11 Keble Road, Oxford. OX1 3QD. November, 1992.

[BUR77] R.M. Burstall and J. Goguen, *Putting theories together to make specifications.* In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pp. 1045-1058, Cambridge MASS, 1977.

[BUR81] R.M. Burstall and J. Goguen, *An informal introduction to specifications using Clear.* In R.S. Boyer and J.S. Moore (Eds.), *The Correctness Problem in Computer Science*, Academic Press, 1981.

[COH86] B. Cohen, W.T. Harwood and M.I. Jackson, *The Specification of Complex Systems*, Addison-Wesley, 1986.

[CRO95] J. Crow, S. Owre, J. Rushby, N. Shankar, M. Srivas, *A Tutorial Introduction to PVS.* Presented at WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida. April, 1995.

[FEI94a] L.M.G. Feijs, *An overview of the development of COLD.* In D.J. Andrews, J.F. Groote, and C.A. Middleburg (Eds.), *Semantics of Specification Languages (SoSL). Proceedings of the International Workshop on Semantics of Specification Languages, Utrecht, The Netherlands, 25-27 October 1993*, pp. 15-22, Workshops in Computing Series, Springer-Verlag, 1993.

[FEI94b] L.M.G. Feijs, H.B.M. Jonkers, and C.A. Middleburg, *Notations for Software Design.* Springer-Verlag, 1994.

[FUT85] K. Futatsugi, J.A. Goguen, J.P. Jouannaud and J. Meseguer, *Principles of OBJ2.* In *Proceedings of the 12th ACM on Principles of Programming Languages*, pp. 52-66, ACM Press, 1985.

[GOG79] J.A. Goguen and J.J. Tardo, *An introduction to OBJ: a language for writing and testing formal algebraic program specifications.* In *IEEE Proceedings on Specifications of Reliable Software*, pp. 170-189. IEEE Press, 1979.

[GUT78] J.V. Guttag and J.J. Horning, *The algebraic specification of abstract data types.* In *Acta Informatica, 10*, pp. 27-52, 1978.

[GUT85a] J.V. Guttag, J.J. Horning and J.M. Wing, *Larch in Five Easy Pieces.* Digital Equipment Corporation, 1985.

[GUT85b] J.V. Guttag, J.J. Horning and J.M. Wing, *The Larch family of specification languages.* In *IEEE Software, 2* (5), pp. 24-36, 1985.

[HAY87] I.J. Hayes (Ed.), *Specification Case Studies.* Prentice-Hall International Series in Computer Science, 1987.

[HOA85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International Series in Computer Science, 1985.

[HUL93] M.E.C. Hull and P.G. O'Donoghue, *Family relationships between requirements and design specification methods.* Department of Computer Science, University of Ulster, Newtownabbey, County Antrim, BY37 0QB, Northern Ireland, UK. In *The Computer Journal, 36* (2), 1993.

[KIE94] R.B. Kieburtz and J. Lewis, *Algebraic Design Language (Preliminary Definition).* Pacific Software Research Centre, Oregon Graduate Institute of Science and Technology, P.O. Box 91000, Portland, Oregon, OR 97291-1000, USA. January, 1994.

[LAV94] G.R.R. de Lavalette, *The static part of the design language COLD-K.* In D.J. Andrews, J.F. Groote, and C.A. Middleburg (Eds.), *Semantics of Specification Languages (SoSL). Proceedings of the International Workshop on Semantics of Specification Languages, Utrecht, The Netherlands, 25-27 October 1993*, pp. 51-82, Workshops in Computing Series, Springer-Verlag, 1993.

[LIS79] B. Liskov and V. Berzins, *An Appraisal of Program Specifications.* In P. Wegner (Ed.), *Research Directions in Software Technology*, Cambridge, Mass: MIT Press, pp. 276-301, 1979.

[LOG90a] L. Logrippo, T. Melanchuck, and R.J. DuWors, *The Algebraic Specification Language LOTOS: An Industrial Experience.* In M. Moriconi (Ed.), *Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development*, pp. 59-66, Napa, 1990.

[LOG90b] L. Logrippo, M. Faci, and M. Haj-Hussein, *An Introduction to LOTOS: Learning by Examples.* University of Ottowa, Protocols Research Group, Ottowa, Ontario, 1990.

[LUC90] D. Luckham, *Programming With Specifications. An Introduction to ANNA, A Language for Specifying Ada Programs.* Springer-Verlag, 1990.

[MIL85] G.J. Milne, *CIRCAL and the Representation of Communication, Concurrency and Time.* In *ACM Transactions on Programming Languages and Systems, 7* (2): pp. 270-298. April, 1985.

[MIL89] R. Milner, *Communicating and Concurrent Systems.* Prentice-Hall International Series in Computer Science, 1989.

[MON91] B. Monahan and R. Shaw, *Model-Based Specifications.* In J.A. McDermid (Ed.), *Software Engineer's Reference Book*, Butterworth-Heinemann, 1991.

[MOR90] C.C. Morgan, *Programming From Specifications.* Prentice-Hall International Series in Computer Science, 1990.

[OU90] The Open University, *CSP Specification.* Topics in Software Engineering Series, Open University, 1990.

[OWR92] S. Owre, J.M. Rushby and N. Shankar, *PVS: A prototype verification system.* In D. Kapur (Ed.), *Automated Deduction - CADE-11*, Springer-Verlag, LNAI 607, pp. 748-752, 1992.

[SAN91] D. Sannella, *Formal program development in Extended ML for the working programmer.* In *Proceedings of the Third BCS/FACS Workshop on Refinement*, Workshops in Computing Series, pp. 99-130, Springer-Verlag, 1991.

[SAN93] D. Sannella, *A survey of formal software development methods.* In A. McGettrick and R. Thayer (Eds.), *Software Engineering: A European Prospective*, pp. 281-297, IEEE Computer Society Press, 1993.

[SPI92] J.M. Spivey, *The Z Notation: A Reference Manual.* Second Edition, International Series in Computer Series, Prentice-Hall, 1992.

[SUN82] C.A. Sunshine, D.H. Thompson, R.W. Erickson, S.L. Gerhart, and D. Schwabe, *Specification and verification of communication protocols in AFFIRM using state transition models.* In *IEEE Transactions on Software Engineering, SE-8* (5). September, 1982.

[WOO88] J. Woodcock and M. Loomes, *Software Engineering Mathematics.* Addison-Wesley, 1988.

[YOU89] W.D. Young, *Comparing Specifications Paradigms: Gypsy and Z*, Technical Report 45, Computational Logic Incorporated, Austin, Texas 78703. June, 1989.

[YUA85] T. Yuasa and R. Nakajima, *IOTA: A modular programming system.* In IEEE Transactions on Software Engineering, *SE-11* (2): pp. 179-187. February, 1985.