Formal Design Methods

Susan Stepney MA PhD

Principal Research Scientist
GEC-Marconi Research Centre

Contents

1 The Rôle and Value of Formal Design Methods in Software Development

With the increasing complexity and scope of applications, particularly in the area of safety critical systems, increasingly sophisticated methods are needed to control and structure software.

The complexity of a system grows faster than its size. Methods of coping with this complexity are needed. Structuring methods were first evolved to cope with the programming (assemblers, then high level languages, then structured languages), then the design phase (structured design and "Hoare logic"), and now the specification (formal specifications and development).

1.1 Why Use Formal Specifications?

The earlier a mistake is made in the software development cycle, the more expensive it is to correct later. A formal specification can highlight problem areas, provide a solid basis for understanding the requirements and, since it is mathematical, allow **reasoning** about the specification. Formal specification increases the effort in the early parts of the lifecycle, with the aim of drastically reducing the effort in the later parts, especially debugging and maintenance.

Formal development from the specification to the software increases confidence that the latter is both bug-free and faithfully implements the former. Maintenance (both debugging and enhancing the specification) is cheaper if there is a rigorous specification of what the software actually does.

Anyone who doubts that a natural language, such as English, is totally unsuitable for rigorously and unambiguously specifying even a simple problem should read Meyer's [60] entertaining account of the repeated failure of attempts to use English to specify a seemingly trivial problem. He identifies the "Seven Sins of the Specifier": noise, silence, over-specification, contradiction, ambiguity, forward reference and wishful thinking. He shows how formalism helps in avoiding these problems.

1.2 What is a Formal Development Method?

A full formal development method provides the following

- **formal specification language :** for specifying a system precisely, so that there is no ambiguity, and abstractly, so that there is a minimum of cluttering detail.
- **refinement rules :** for refining the formal specification to a program, adding the required implementation detail in a controlled and rigorous manner.
- **methodology :** for developing the specification from the requirements, and for refining it down to an implementation.
- **tool support :** to provide automated support for each stage of the specification and development process.

Each of these terms is described in more detail below.

1.2.1 Specification

A formal specification language is a mathematical notation designed for specifying systems. Being mathematical, the language has formal rules for manipulating statements in the language and for proving theorems.

The formal notation need not be restricted to specifying software. For example, it is often advantageous to specify the environment as well as the (hardware and software) system. This highlights the assumptions being made about the environment, and is useful when simulating it for validation and testing.

A **formal specification** is a restatement of an (informal) requirements specification in a formal specification language. The formal specification should form an unambiguous basis for the contract between the customer and supplier. It should also form the starting point against which implementations are verified, and for any future modifications or upgrades to the requirements. By definition, it is impossible to prove a formal system equivalent to an informal system. Hence the formal specification has to be *validated* against the customer's informal (and sometimes unstated) requirements.

1.2.2 Refinement

The high level specification talks in the language of mathematics, and may manipulate objects such as sets and sequences that are not provided in the lower level implementation language. A set may be implemented as an array, or a linked list, or by some other means, depending on the application.

Refinement is the processes of successively changing a specification from one version to another which is closer to the implementation. This process continues until the specification can be mapped directly into the chosen programming language. It should be possible to verify every refinement step, showing that any implementation of the refined specification will also meet the specification from which it was refined.

The refinement method provides **proof obligations**. These are a set of rules to generate the proofs that are required to show that a particular implementation choice is valid. The refinement method is rarely used in formal specifications, being tedious and difficult. This is in part due to a lack of tool support for the process.

1.2.3 Methodology

Most current formal "methods" are simply formal *languages*, sometimes with refinement rules. Very few contain a methodology for developing and refining specifications. One recent attempt to remedy this problem is the Forest project (see later).

1.2.4 Tool Support

Automated tool support is valuable at each level of the design process, when writing and checking the specification, and during the refinement process. Current formal methods provide this support to a greater or (usually) lesser degree. [78] describes some of the tools being built.

Initially, the main problem with writing most formal specifications is getting machine support for all their mathematical symbols. TEX [54], and in particular LATEX [55], are becoming popular for typesetting specifications. There are sets of special purpose LATEX macros for Z and VDM now available. Various toolsets come with their own special editors that can display the necessary symbols directly on high-resolution workstation screens.

An important component of a fully mature method is a theorem prover. This is required for proving the correct refinement of non-trivial systems. These proofs can, in principle, be done by hand, but they are long, tedious, and error-prone.

One area of current research is that of executable specifications. There are two opposing opinions in this area. One opinion is that executing the specification is a powerful and valuable aid to understanding the problem and debugging the specification. The other opinion can be summarized as: *if you can execute it, it isn't a specification.* This is because executable specifications are at a lower level of abstraction than non-executable ones (see the following sections on abstraction and non-determinism for further arguments). The two bodies of opinion differ on whether the gain of executability makes up for the loss of abstraction.

1.3 Proofs and Theorems

One of the most powerful properties of a formal method is the reasoning that can be done at various stages. At the higher levels, desirable properties of the system can be posed as theorems, and so proved to be consequences of the specification. Also, possibly more important, undesirable properties can be proved *not* to be properties of the specification. In fact, Woodcock [81] claims: *the presence of these theorems and their proofs distinguishes better specifications from poorer ones.* Because these proofs take place early in the lifecycle, mistakes in the specification can be found before too much effort has been involved in the development.

1.3.1 Formal Proof and Rigorous Argument

What a mathematician thinks of as a proof, and what a computer can produce or check, are very different things. Jones [52, pp282&287] contrasts the following

- A **formal proof** is one in which all steps are mechanical; thus a formal proof can be checked by a computer program.
- A **rigorous argument** outlines how a proof could be constructed; the reason for accepting such an argument is the knowledge of how it could be made formal.

According to this definition most mathematical proofs in the literature are in fact rigorous arguments. In fact, any non-trivial formal proof is so long, intricate and, above all, dull, that it would almost certainly have to be derived, as well as checked, by a computer. For a thought provoking counter-argument to a reliance on formal proofs, see [20] for a discussion of the philosophical differences between mathematical and computer generated proofs.

1.3.2 Validation and Verification

The terms **validation** and **verification** occur frequently in the context of formal (as well as informal) system development. They are different processes. The following definitions are taken from [16, p97]:

- **verification** involves rigorous, formally justifiable proofs; for example, that a design satisfies a specification or that a specification is internally consistent (i.e. non-vacuous).
- **validation** involves convincing demonstrations of the soundness of conjectures; for example, that a formal specification 'captures' a system requirement or that an implementation is acceptable to the customer.

So verification is *proof* whereas validation is *testing*. A specification is a mathematical model of part of the real world. Properties of this model can be formally derived and verified, but to show that these are also properties of the real world requires validation. Dijkstra's well-known comment that *program testing can be used to show the presence of bugs, never to show their absence* applies equally to specification validation.

1.4 Abstraction: What, not How

The question is often asked: why not use the same language for specification and implementation? A specification language is designed for stating the problem and for reasoning about the problem; a programming language is designed for stating the solution, or implementation, of the problem (and, very rarely, for reasoning about the solution). These two, problem and solution, are at different levels of abstraction, and currently require different languages. Work is being done on bridging the gap (one notable area being functional languages) but this is still a research topic.

One important way in which specifications differ from implementations is that they are non-algorithmic – they say *what* should be done, not *how* it should be done. For example, a specification for sorted data will merely say that the result is the data in some particular order, not that this order should be achieved by a particular sorting algorithm. *Any* implementation which meets the specification can be used; the most appropriate one may be chosen to satisfy other, non-functional, requirements (performance, storage space, and, all else being equal, ease of implementation).

1.5 Non-determinism

Specifications can exhibit either, or both, of **under-specification** and **non-determinism**. In each case, the result of an operation is not specified fully, only constrained, but in a different way:

- **Under-specification :** Once chosen, the operation must always give the same result each time it is executed. This is sometimes called a **loose** specification. For example, consider a hash function – it might not be important which particular function is used, but it must always give the same answer in a given implementation.

- **Non-determinism :** The operation can give a different result each time it is executed. For example, consider a storage manager – it can give different answers (different blocks of memory) on different calls.

An implicit specification is one where the result of an operation is given in terms of some conditions the result must satisfy, as contrasted with an explicit result given in terms of a formula. Implicit specifications are often at a higher level of abstraction, and can be easier to understand, than explicit ones. A consequence of implicit specifications, however, is **unbounded non-determinism**, which results in a specification being non-executable.

Note that refining a specification can reduce the amount of non-determinism. All the non-determinism has to be refined out before the final implementation step to produce a deterministic system.

Spivey [73, §5.4] gives examples of loose and non-deterministic specifications, and argues that they are necessary properties of a specification method, even if only deterministic systems are to be specified.

1.6 Disadvantages of Using Formal Methods

Formal methods do not provide a magic solution whereby all the problems of specifying and developing correct, reliable software will evaporate overnight.

One of the oft quoted disadvantages of formal methods is that they are hard work, both to learn, and to use. Actually, the syntax of a formal method is probably no harder to learn that that of one of the less verbose computing languages (for example, C or APL). It is certainly hard work to use them, but that is mainly because they are being applied to intrinsically difficult problems; *any* method would be hard work!

The current lack of automated tool support certainly does not help to endear the use of formal methods. However, it should be realised that a formalism in itself is a tool, a "tool for thought", and can be valuable even without any computer-based support. It might even be argued that the very lack of support encourages a high level of abstraction and conciseness!

The major drawback of formal methods is that they cannot (yet) be used to specify all the requirements of a system. They are still weak in the area of real-time constraints. Also, safety properties such as fault tolerance and reliability are not covered. One reason for this is that formal methods deal with a model of the real world, and safety problems usually occur at just those points where the model breaks down.

All these shortcomings are subjects of current research, and so the situation should improve with time. Currently, given that formal methods cannot do everything, probably the best approach to take is to use them as much as possible, where appropriate. Consider the analogy with the choice between using a high level language like Pascal or using an assembler: the former cannot do everything, but that does not mean all applications should be written exclusively in the latter.

2 Mathematical Background

There follows a brief description of the sort of mathematical background required to use formal specification languages. There are a variety of textbooks that cover these areas. For

the more mathematically inclined, [4,32,33,34] are typical introductions. For discussions geared towards program specification, see [23,83].

2.1 Some Terminology

A **calculus** is a collection of rules for calculating with symbols. Propositional calculus is used to determine the truth or falsehood of propositions, using logical connectives (**and, or, not, implies**). Predicate calculus extends this by allowing variables and quantifiers (**there exists, for all**). Lambda calculus is used for defining and manipulating functions.

An **algebra** is a set, along with a collection of operations to manipulate and combine members of the set. An example of an algebra is the set of strings and the associated string operations.

A **logic** is a study of the principles of valid inference. It consists of a set of statements, and a set of rules for constructing new statements. The statements make assertions about the state of a reference world. A **theory** is a particular collection of statements. If these statements are all true in the reference world, then the world is a **model** of the theory. The rules for constructing new statements define the valid inferences of the logic.

- **First-order logic** provides the means to make assertions about *the* state of the world. Its statements are constructed using first-order predicate calculus ("first-order" means that the allowed variables to predicates cannot themselves be predicates). First-order logic implicitly assumes that the world is fixed and static.
- **Modal logic** allows for variation and variability in the world. The world state is assumed to be any one of a set of distinct possible world states, and the theory associated with each world state can make assertions about the theory applicable to other possible related states.
- **Higher order logics** allow the variables in predicates to be functions. This provides a more powerful, if more complicated, reasoning system.

2.2 Propositional Calculus

A proposition p is a mathematical statement that has a truth value, either true ($p = T$) or false ($p = F$). Compound propositions can be constructed from simpler ones using logical connectives.

The **negation** of proposition p is written $\neg\ p$, "not p". The **logical disjunction**, "inclusive or", of the propositions "p or q" is written $p \vee q$. The truth values of these constructs are *defined* by the truth tables:

p	q	$\neg\ p$	$p \vee q$
T	T	F	T
T	F		T
F	T	T	T
F	F		F

Other connectives are defined for readability.

- **logical conjunction :** p and q, $p \wedge q$
- **logical implication :** p implies q, $p \Rightarrow q$
- **logical equivalence :** p if and only if q, $p \Leftrightarrow q$

All these others can all be defined in terms of "not" and "or":

$$p \wedge q \equiv \neg (\neg p \vee \neg q)$$
$$p \Rightarrow q \equiv \neg p \vee q$$
$$p \Leftrightarrow q \equiv p \Rightarrow q \wedge q \Rightarrow p$$

The corresponding truth tables are:

p	q	$p \wedge q$	$p \Rightarrow q$	$p \Leftrightarrow q$
T	T	T	T	T
T	F	F	F	F
F	T	F	T	F
F	F	F	T	T

In principle, the truth or falsity of a proposition can be found by producing the relevant truth table. However, since a proposition with n components has a truth table with 2^n lines, this can become tedious. So there are various laws which allow compound propositions to be manipulated, in order to simplify them or convert them to some standard form:

- **identity :** $p \vee F \equiv p$, and $p \wedge T \equiv p$
- **idempotency :** $p \vee p \equiv p$, and $p \wedge p \equiv p$
- **double negation :** $\neg \neg p \equiv p$
- **distributive laws :** $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$, and $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
- **de Morgan's laws :** $\neg (p \vee q) \equiv \neg p \wedge \neg q$, and $\neg (p \wedge q) \equiv \neg p \vee \neg q$

Certain types of proposition are distinguished:

- **tautology :** Such a proposition is always true, irrespective of the truth values of its components. For example, $p \vee \neg p$.
- **contingency :** The truth value of such a proposition depends on (is contingent on) the truth values of its components. For example, $p \vee q$.
- **contradiction :** Such a proposition is always false, irrespective of the truth values of its components. For example, $p \wedge \neg p$.

The main rule of inference used in logical proofs is **modus ponens**: if $p \Rightarrow q$ is true, and if p is true, the the inference is that q is true. This is customarily written in the form

$$\frac{p \Rightarrow q, \quad p}{q}$$

The convention is, given the truth of the proposition(s) above the line, the truth of the proposition(s) below the line can be inferred.

Other rules of inference include the rule that if $p \Rightarrow q$ is true, but q is not true, the inference is that p is not true:

$$\frac{p \Rightarrow q, \quad \neg\, q}{\neg\, p}$$

and the transitive rule

$$\frac{p \Rightarrow q, \quad q \Rightarrow r}{p \Rightarrow r}$$

The most common indirect proof method is **reductio ad absurdum**. In order to prove a proposition p, first assume its negation, $\neg\ p$, and use this assumption to derive a contradiction. This contradiction implies that the assumption is not true, and hence p is true.

For a wider range of laws and inference rules, see any standard textbook on logic, *e.g.* [4,33].

2.3 Predicate Calculus

Propositional calculus is not sufficiently powerful for the needs of specifying programs. Predicate calculus extends propositional calculus by allowing variables. A predicate P is a statement describing a property that a variable x may or may not have. When a particular value is substituted for the variable, the predicate becomes a proposition with a truth value. It is assumed that there is some *universe of discourse* from which values for x are drawn, such that the propositions formed from $P(x)$ are always either true or false.

Using predicate calculus, it is possible to make statements about some, or all, of the members of a set, by using **quantifiers**.

The **existential quantifier** gives a way of saying that there exists some x, such that $P(x)$ is a true proposition. (This statement could, of course, itself be false!) It is written $\exists x \bullet P(x)$, and read "there exists an x such that $P(x)$", or alternatively, "for some x, $P(x)$". If there are a finite number of values for x in the universe of discourse, $x_1 \ldots x_n$, the existential quantifier can be replaced by logical disjunction:

$$\exists x \bullet P(x) = P(x_1) \vee P(x_2) \vee \ldots \vee P(x_n)$$

The **universal quantifier** gives a way of saying that $P(x)$ is true no matter what x is substituted in P. It is written $\forall x \bullet P(x)$, and read "for all x, $P(x)$". If there are a finite number of values for x, the universal quantifier can be replaced by logical conjunction:

$$\forall x \bullet P(x) = P(x_1) \wedge P(x_2) \wedge \ldots \wedge P(x_n)$$

The two quantifiers are related by

$$\forall x \bullet P(x) \equiv \neg \exists x \bullet \neg P(x)$$

(if $P(x)$ is true for all x, then there is no x for which $\neg P(x)$ is true).

There are inference rules for quantifiers. For example, the rule of $\forall$-elimination is

$$\frac{a, \quad \forall x \bullet P(x)}{P(a)}$$

This is the general rule for the classic example, *Socrates is a man, all men are mortal, hence Socrates is mortal.* Another inference rule is

$$\frac{P(a)}{\exists x \bullet P(x)}$$

This is a **constructive** proof: in order to prove there exists an x for which P is true, produce such an x. There is also a **non-constructive** proof, which involves demonstrating that $\neg \exists x \bullet P(x)$ leads to a contradiction. This proof gives no indication of which particular value(s) of x do actually make P true.

2.4 Set Theory

Sets can be used to define new variable types and program states. A set can be thought of as a collection of distinct objects, its elements. These elements can be (virtually) anything, including other sets.

Sets can be written by explicitly listing all their elements, for example, $\{a, b, c\}$. The order in which the elements are written is unimportant.

A property can be used to specify a set, using the notation

$$\{x \mid P(x)\}$$

which means the set of all x for which $P(x)$ is true. This method of specifying a set is called **set comprehension**. The assumption that properties can be used to specify sets in this way is one of the bases of axiomatic set theory, and is known as the **axiom of replacement**. If the universe of discourse W, the set from which the x are drawn, is not obvious from context, it must be specified:

$$\{x: W \mid P(x)\}$$

There has to be some universe of discourse; there can be no such thing as "the set of all sets" in axiomatic set theory. It was the failure to realise this fact in pre-axiomatic set theory that lead to the infamous **Russell paradox**, usually summarized as *there is a barber who shaves everyone who does not shave himself; who shaves the barber?* Axiomatic set theory solves the problem: there is no such barber.

- **set membership :** $x \in S$
  "x is an element of S", "x is in S".

- **set equality :** $A = B$
  Two sets are equal if and only if they have the same elements (this is the **axiom of extensionality** in axiomatic set theory): $A = B \Leftrightarrow (\forall x \bullet x \in A \Leftrightarrow x \in B)$.

- **empty set :** $\{\}$
  The set with no members: $\neg\, \exists x \bullet x \in \{\}$. The axiom of extensionality implies that the empty set is unique.

- **subset :** $A \subseteq B$
  "A is a subset of B". A subset is a set formed from some or all of the elements of another: $A \subseteq B \Leftrightarrow (\forall x \bullet x \in A \Rightarrow x \in B)$. All sets have the empty set and themselves as subsets: $\{\} \subseteq A, A \subseteq A$. If A and B are subsets of each other, they are equal: $A \subseteq B \wedge B \subseteq A \Leftrightarrow A = B$; this fact is often used as the basis for a proof of set equality.

- **proper subset :** $A \subset B$
  A subset that is not the set itself: $A \subset B \Leftrightarrow A \subseteq B \wedge A \neq B$

- **union :** $A \cup B$
  The union of two sets A and B is the set containing all elements that are in A or B or both: $A \cup B = \{x \mid x \in A \vee x \in B\}$

- **intersection :** $A \cap B$
  The intersection of two sets A and B is the set containing all elements that are in both A and B: $A \cap B = \{x \mid x \in A \wedge x \in B\}$

- **set difference :** $A - B$
  The difference of two sets A and B is the set containing all elements that are in A but not in B: $A - B = \{x \mid x \in A \wedge x \notin B\}$

- **complement :** A'
  The complement of set A is the set containing all elements that are not in A, defined relative to some universe of discourse W: $A' = W - A$

- **disjoint :** $\mathit{disjoint}(A, B)$
  Two sets are disjoint if they have no elements in common:
  $\mathit{disjoint}(A, B) \Leftrightarrow A \cap B = \{\}$

- **cardinality :** $\#A$
  The number of distinct elements in the set.

- **power set :** $\mathbb{P}\, A$
  The set of all subsets of A: $B \subseteq A \Leftrightarrow B \in \mathbb{P}\, A$

- **ordered pair :** (x, y)
  A pair of elements where the order of writing is important.

- **Cartesian product :** $A \times B$
  Also called cross product, the Cartesian product of A and B is the set of all ordered pairs where the first element is drawn from A and the second from B:
  $A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$

The close correspondences between set union and logical disjunction, and between intersection and logical conjunction, are obvious from their definitions. There is also a correspondence between set inclusion and logical implication:

$$\{x \mid P(x)\} \subseteq \{x \mid Q(x)\} \Leftrightarrow P(x) \Rightarrow Q(x)$$

This property in particular is important in the formal construction of programs.

Set expression can be manipulated using various laws. Unsurprisingly, some of these laws closely resemble the propositional calculus laws. In the following, W is the universe of discourse, and A, B, C are arbitrary subsets of W.

- **identity laws :** $A \cup \{\} = A$, and $A \cap W = A$
- **idempotency :** $A \cup A = A$, and $A \cap A = A$
- **distributive laws :** $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$,
  and $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
- **de Morgan's laws :** $(A \cup B)' = A' \cap B'$, and $(A \cap B)' = A' \cup B'$
- **power set size :** $\#(\mathbb{P}\, A) = 2^{\#A}$

For a wider range of laws, see any standard textbook on set theory, *e.g.* [32,34].

2.5 Relations

A relation R between two sets A and B relates elements of A to elements of B. Each subset of $A \times B$ defines a different relation, and so the set of all possible relations from A to B is $\mathbb{P}(A \times B)$. If R is particular subset of $A \times B$, and R relates $a \in A$ to $b \in B$, then (a, b) is an element of R. The following terminology is used for relations:

- $R: A \leftrightarrow B \Leftrightarrow R \in \mathbb{P}(A \times B)$.
- $a\ R\ b \Leftrightarrow (a, b) \in R$.
- $a \mapsto b \Leftrightarrow (a, b) \in R$.

Any particular R need not map all elements of A to something. The set of those it does map is called the **domain** of the relation. Similarly, the subset of B mapped to by R is called the **range**.

$$dom\ R = \{a \in A \mid \exists b \in B \bullet a\ R\ b\}$$
$$ran\ R = \{b \in B \mid \exists a \in A \bullet a\ R\ b\}$$

The inverse of relation a relation R, written R^{-1}, reverses the ordered pairs:

$$R^{-1} = \{(b, a) \mid a\ R\ b\}$$

The sets A and B do not have to be distinct. It is possible to have a relation on a single set: $R \in \mathbb{P}(A \times A)$. Such relations may possess various symmetry properties:

- **reflexive :** every element is related to itself: $\forall a \in A \bullet a \; R \; a$
- **irreflexive :** no element is related to itself: $\forall a \in A \bullet \neg (a \; R \; a)$
- **symmetric :** if a is related to b, then b is related to a, hence $R = R^{-1}$: $\forall a, b \in A \bullet a \; R \; b \Leftrightarrow b \; R \; a$,
- **asymmetric :** if a is related to b, then b is not related to a: $\forall a, b \in A \bullet a \; R \; b \Rightarrow \neg (b \; R \; a)$
- **antisymmetric :** if a is related to b and b is related to a, then a equals b: $\forall a, b \in A \bullet a \; R \; b \wedge b \; R \; a \Rightarrow a = b$
- **transitive :** if a is related to b and b is related to c, then a is related to c: $\forall a, b, c \in A \bullet a \; R \; b \wedge b \; R \; c \Rightarrow a \; R \; c$

An **equivalence relation** is one that is reflexive, symmetric and transitive.

Sometimes it is convenient to use a relation that is known to have one of these symmetry properties. The transitive (similarly reflexive, or symmetric) **closure** of a relation R is the smallest transitive (reflexive, or symmetric) relation that includes R as a subset. The transitive closure is often written as R^+, and the reflexive transitive closure as R^*.

2.6 Functions

A function is a special kind of relation: each element of the domain maps to a unique element of the range. More than one element may map to the same result, however. A function is a **many-to-one** relation. A different notation is used to denote functions; if f is a function from A to B, and if x is in the domain of f, then the (unique) element x is mapped to by f is written $f \; x$.

$$f \; x = y \equiv (x, y) \in f$$

So a function can be defined by

$$\forall a \in A; \; b_1, b_2 \in B \bullet f \; a = b_1 \wedge f \; a = b_2 \Rightarrow b_1 = b_2$$

The function $f \in \mathbb{P}(A \times B)$ is written $f : A \rightarrow B$.

The inverse of a function is not necessarily a function; at its most general it is a **one-to-many** relation.

Various special types of function are distinguished. In the following, f is a function from A to B, *i.e.* $f : A \rightarrow B$.

- **partial function :** the domain of f is not the whole of A: $dom \; f \subset A$
- **total function :** the domain of f is the whole of A: $dom \; f = A$
- **surjection :** the range of f is the whole of B: $ran \; f = B$. In this case f is said to map $dom \; f$ "onto" B.

- **injection :** each element of the domain is mapped to a different element of the range (**one-to-one**): $f\ a_1 = f\ a_2 \Leftrightarrow a_1 = a_2$. The inverse of an injective function is also a function.
- **bijection :** a function that is total, one-to-one and onto. Consequently A and B have the same size: $\#A = \#B$. A bijection can be thought of as a relabelling of the elements of A by the elements of B.
- **identity :** the function that maps each element to itself: $\forall a \in A \bullet id\ a = a$.

Functions can be combined by applying a function to an argument, then applying another function to the result. This is called **functional composition**. If $f: A \rightarrow B$ and $g: B \rightarrow C$, the composite of g and f is the function $g \circ f: A \rightarrow C$ and is defined by

$$\forall a \in A \bullet (g \circ f)(a) = g(f(a))$$

2.7 Sequences

Sequences are a very useful construct in specifications. The order of the elements in a sequence is important. Sequences can be defined in terms of functions, from an initial segment of the natural numbers to the given set. So a sequence of n elements from A is $s = \{1 \mapsto a_1, 2 \mapsto a_2, \ldots, n \mapsto a_n\}$. This is usually abbreviated to $s = [a_1, a_2, \ldots, a_n]$. This sequence has length $\#s = n$. Particular elements of the sequence can be extracted using the fact that it is a function: $s(m) = a_m$.

Sequences can be joined together, by **concatenation**. Operations like **head**, **tail** and **rev** can be defined in terms of functions.

$$\begin{array}{ll} [a, b] \frown [c, d, e] & = [a, b, c, d, e] \\ head[a, b, c] & = a \\ tail[a, b, c] & = [b, c] \\ rev[a, b, c] & = [c, b, a] \end{array}$$

2.8 Proof by Induction

As well as the proof techniques sketched in the sections of predicate calculus and set theory, the technique of **proof by induction** is a powerful tool.

In its best known form, it uses induction over natural numbers. Given a predicate defined on natural numbers, if the predicate is true for some base number b, and if, assuming it is true for an arbitrary $k \geq b$ implies that it is true for $k + 1$, then the inference is that it is true for all $n \geq b$:

$$\frac{P(b), \quad \forall k \geq b \bullet P(k) \Rightarrow P(k+1)}{\forall n \geq b \bullet P(n)}$$

So a proof by induction involves two steps: proof of $P(b)$ (the **base step**), and proof of $P(k) \Rightarrow P(k + 1)$ (the **induction step**).

The validity of the inference rests on the fact that the natural numbers are themselves defined "inductively": there is a base number (zero) and a successor function that generates the rest from the base. Data structures which have an analogous form, for example sequences and binary trees, have analogous induction proof mechanisms. For example, in the case of a binary tree, the idea is to prove a proposition true for the null tree (leaf), and to show that if it is true for a left sub-tree and right sub-tree, then it is true for the larger tree made from the sub-trees. From this the inference is that the proposition is true for all binary trees. So, inventing some notation purely for the purpose of exposition:

$$\frac{P(leaf), \quad \forall t_1, t_2 : TREE \bullet P(t_1) \wedge P(t_2) \Rightarrow P(tree(t_1, t_2))}{\forall t : TREE \bullet P(t)}$$

2.9 Alternative Notations

It can sometimes seem that every author uses a different notation for logical connectives, quantifiers, sets, *etc.* Some of the more common alternative notations are given below:

- logical negation : $\neg\, p$, $\sim p$
- logical conjunction : $p \wedge q$, $p \& q$, $p.q$
- logical disjunction : $p \vee q$, $p \mid q$
- logical implication : $p \Rightarrow q$, $p \subset q$, $p \rightarrow q$
- logical equivalence : $p \Leftrightarrow q$, $p \leftrightarrow q$
- existential quantifier : $\exists x \bullet P(x)$, $(\exists x)(Px)$, $\mathsf{E}x.P(x)$
- universal quantifier : $\forall x \bullet P(x)$, $(\forall x)(Px)$, $\mathsf{A}x.P(x)$
- empty set : $\{\}$, $\varnothing$
- set difference : $A - B$, $A \setminus B$
- cardinality : $\#A$, $card(A)$
- sequence : $[a_1, a_2, \ldots]$, $< a_1, a_2, \ldots >$

3 Specification Styles

There are many different formal specification languages, and many different styles that a formal specification can be written in (*e.g.*, algebraic, model based, functional). Often, a particular language can be used for a range of styles.

3.1 Abstract Data Type Approach

An Abstract Data Type (ADT) is a package containing both a data type and those operations which can act on the data. Simple examples of ADTs are a stack (with operations like *push* and *pop*) and a sequence (with operations like *head* and *append*). When the ADT approach is used for a programming language, it is called an **object-oriented** language, the data types are called classes, and the operations are called methods.

There are two main styles for defining ADTs:

- **algebraic**, also called axiomatic, or implicit.
- **model-based**, also called explicit.

Here **implicit** and **explicit** refer to the system's state, rather than how the result of an operation is specified.

3.1.1 Algebraic, or Axiomatic Approach

With the axiomatic approach, the ADT's operations are defined in terms of equations (axioms). For example, a stack specification in implicit, axiomatic style could have its operations defined by equations like:

$$pop(push(elem, stack)) = stack$$
$$top(push(elem, stack)) = elem$$

The first axiom says that if you *push* an *elem* onto a *stack*, then *pop* it off again, the result is the original *stack*. The second says that if you *push* an *elem* onto a *stack*, that is the *top* element.

The specification is built up from a set of these equations to define all the allowed operations.

Complicated expressions can be simplified by treating the equations as **rewrite rules**. They are systematically substituted until a non-reducible expression occurs. This is taken to be the result.

OBJ is an example of an algebraic specification language.

3.1.2 Model Based Approach

With the model based approach, the state of the system is *explicitly* modelled. Operations are defined which specify the required changes of the system state as a set of equations relating the **before** and **after** states. For example, in a VDM-like notation, the *PUSH* operation for a *Stack* specification could be defined like

$$Stack = \text{seq of } X$$

$$PUSH \quad (x{:}X)$$
$$\text{ext wr } stk \ : \ Stack$$
$$\text{post } \overleftarrow{stk} = [x] \frown stk$$

This shows a stack being modelled as a sequence of elements of type X. The operation *PUSH* is specified as taking an argument x of type X, and is defined in terms of a post-condition: the after stack $\overleftarrow{stk}$ is the same as the before stack *stk* with x added to the beginning. (Note here the potential for overspecification – there is no reason why the operation could not have been modelled as adding x to the other end of the sequence. However, an implementation could always build a model of sequences "the wrong way round", avoiding the problem.)

VDM and Z are example languages which can be used to specify ADTs in a model based style (although they do not have explicit support for inheritance, which is required for a full ADT approach). They can also be used in other ways. For example Jones [52, §9.2] shows how to use VDM in an axiomatic style to specify a stack.

3.2 Process Based Approach

The previously mentioned techniques tend to be used for sequential systems. Interacting and distributed systems are even more complex to specify, design, implement, and reason about.

The process based methods derive from work on a mathematical theory of communicating processes based on a synchronous message passing paradigm. In particular, they address system synthesis and concurrency issues.

A system comprises several independent, concurrent processes which communicate with one another by means of synchronous message passing over specified data channels. Each process has a well defined set of rendezvous states where it waits until it is able to exchange a message with some other process. Asynchronous messages can be modelled by adding suitable buffering to the communications channel.

A new error that can easily arise when specifying a parallel, process based system is **deadlock**: although every process is willing to engage in further communication, they cannot agree on what it should be, and so nothing further happens. There is also the possibility of **livelock** (analogous to an infinite loop in the sequential case), where processes engage in infinite internal communication, without doing anything useful. So a process based system has extra proof obligations: to prove freedom from deadlock and livelock.

Example languages that can be used in the process based approach are CCS, CSP and LOTOS.

3.3 Lambda Calculus and Functional Languages

Lambda calculus [40,41] provides a way of defining a function without having to give it a name, by using a **lambda-abstraction**. A lambda-abstraction can be defined at its point of use. For example, instead of defining

$$f(x) = x^2 + 1$$
$$f(3) = 10$$

the same function can be defined by

$$\lambda x \bullet x^2 + 1$$
$$(\lambda x \bullet x^2 + 1)(3) = 10$$

Functions of several variables can be defined, and supplying *some* arguments results in a function of fewer variables. For example:

$$\lambda y \bullet \lambda x \bullet x^2 + y^2$$
$$(\lambda y \bullet \lambda x \bullet x^2 + y^2)(3) = \lambda x \bullet x^2 + 9$$

Lambda calculus also provides rules for combining, manipulating and simplifying functions.

Lambda calculus is one method that can be used to define functions in languages like Z and VDM. It is the way that all functions are defined in functional programming languages; it has been said that any *pure* functional language is merely syntactic sugar for lambda-calculus (*typed* lambda-calculus is used for some functional languages). However, most functional languages have a greater or lesser degree of impurity in them, which to a correspondingly lesser or greater degree makes them difficult to reason about. Even so, pure functional languages are at a sufficiently high level of abstraction, and are sufficiently amenable to mathematical manipulations, that they can also be used for specification.

Lisp is the original, and probably least pure, of the functional languages. It comes in a variety of dialects. ML [80,35] is a very popular functional language; it is used to implement many of the support tools for other formal methods. Other functional languages include SASL, Hope [11] and Miranda [77].

3.4 Temporal Logic Approaches

Temporal logics describe the temporal properties of systems by using ideas and notations from modal logic. A temporal logic specification identifies the system's state at a set of discrete time intervals, and specifies how properties of the state during one interval depend on the properties of the state during other intervals, using operators such as **always, next, sometimes, eventually, while** and **until** (the exact choice of operators depends on the particular formulation of the logic). The logic has laws such as

$$\textit{sometimes}\ P \Leftrightarrow \neg\ \textit{always} \neg\ P$$

Manipulations and proof rules follow in a way analogous to classical logic.

See [2] for a specification of a lift system in a temporal logic. [67] describes an executable temporal logic. Real-time extensions to temporal logics are being researched.

4 Existing Methods

Formal specification is still in its infancy. New methods crop up regularly, and tool support also increases.

4.1 OBJ

OBJ [27,26] is a high-level language used for writing algebraic specifications.

In OBJ a specification is partitioned into ADTs called 'objects'. An object can import other objects that it wants to use, and objects can be instantiated from generic objects. For example, an OBJ definition of a *List* might look like:

$$
\begin{array}{l}
\textit{OBJ List/Element} \\
\textit{SORTS list} \\
\textit{OPS} \\
\quad \textit{new}: \rightarrow \textit{list} \\
\quad \textit{add}: \textit{elem list} \rightarrow \textit{list} \\
\quad \textit{join}: \textit{list list} \rightarrow \textit{list} \\
\textit{VARS} \\
\quad e: \textit{elem} \\
\quad l, l': \textit{list} \\
\textit{EQNS} \\
\quad (\textit{join}(\textit{new}, l) = l) \\
\quad (\textit{join}(\textit{add}(e, l), l') = \textit{add}(e, \textit{join}(l, l'))) \\
\textit{JBO}
\end{array}
$$

This defines an object *List*, which references another object *Item*. The declaration *SORTS list* declares the relevant data type. The *OPS* keyword introduces the various operations that can be performed on a *list*, along with their types. *VARS* defines some internal variables, and *EQNS* uses these variables in various equations that implicitly define the operations.

UMIST OBJ [18,25] is an executable subset of the full OBJ language which permits the execution, testing and animation of OBJ specifications. With UMIST OBJ, a specification can be checked and compiled, and then particular expressions can be evaluated.

4.2 Z Specification Language

Z [74] is based on typed set theory and first-order predicate calculus. In Z, a type is a **maximal set**: values may belong to just one type [81] (contrast this with VDM's subtypes). Z extends this mathematics with **schemas**, a way of structuring the formal text. The **schema calculus** gives rules for combining schemas.

A good Z specification should consist of a mix of formal mathematics and informal explanatory text. The formal part provides the precise, unambiguous definition of the system (the "what"). The informal part acts as a commentary (the "why"); it can be consulted to find out what aspects of the real world are being described.

Z specifications can be written in different styles, for example:

- **model based :** The specification describes the state of the system, and the various operations that can change the state. Transaction processing systems often fall naturally into this category.
- **functional :** A function that transforms the input into the output is specified as a combination of simpler functions. For example, a compiler can be specified as a function that takes source code as its input and produced object code as its output.

- **object-oriented :** This is a more recent approach, allowing schemas to be inherited, rather than simply included, by other schemas. It is still the subject of research [12,72].

A specification need not be restricted to one style. For example, some state changes may be specified in a functional manner, as may some of the methods for objects, and objects themselves can have internal state.

Z syntax is described in [53,74], with rather more gentle introductions in [45,59,83]. Z semantics is described in [73]. Schemas are described in [59,81]. Various example specifications are given in [3,37,36,66,75].

4.2.1 Z Toolkit

Z has a rich pre-defined mathematical toolkit, with a variety of objects:

- **sets :** the building block for all the types
- **relations :** defined as sets of ordered pairs
- **functions :** special kinds of relations, with various constraints
- **sequences :** defined in terms of functions
- **bags :** (like sets, except that the number of times an element occurs is important), defined in terms of functions

along with a host of pre-defined operations on these. Quite powerful operations can be built from this toolkit. For example, the number of times a particular element, x, occurs in a sequence, S, is given by $count(items\ S)x$. In this expression, *items* is a function that forms a bag from a sequence, and *count* is a function to count the number of items in a bag. Both are pre-defined operations in Z.

All these operations are defined in Z by using Z itself. The same mechanism is used to define new, special-purpose, operations for each specification. Functions can be defined in a variety of styles. For example, the reverse of a sequence s can be defined explicitly

$$\forall m: 1 \mathinner{..} \#s \bullet (rev\ s)(m) = s(\#s - m + 1)$$

or by a lambda-abstraction

$$rev\ s = \lambda\, n: 1 \mathinner{..} \#s \bullet s(\#s - n + 1)$$

or recursively

$$\begin{aligned} rev <> &= <> \\ rev(s^\frown < e >) &= < e > {}^\frown rev\ s \end{aligned}$$

Libraries can be built up for inclusion in other specifications.

4.2.2 Schemas

An important structuring feature of Z is the **schema**. Schemas are used to define both the states and the operations of ADTs. An operation defines the relation between the states before and after execution, and can have inputs and outputs.

Schemas can be combined together, and this allows modularization and separation of concerns. To give an indication of how this works, a very simple example of a symbol table is given here.

A symbol table can be modelled as a mapping from symbols to values (for the purposes of exposition, this table has a limited size):

$$
\begin{array}{|l}
\hline
SYMBOL\_TABLE \\
st : SYMBOL \pfun VALUE \\
\hline
\# st \leq MaxTableSize \\
\hline
\end{array}
$$

The before state of the symbol table (st) is transformed into the after state (st') by an operation. The conventional Z shorthand for this before and after state is:

$$
\begin{array}{|l}
\hline
\Delta SYMBOL\_TABLE \\
SYMBOL\_TABLE \\
SYMBOL\_TABLE' \\
\hline
\end{array}
$$

Here, two schemas have been **included** in another. They can be expanded to show the inner details:

$$
\begin{array}{|l}
\hline
\Delta SYMBOL\_TABLE \\
st : SYMBOL \pfun VALUE \\
st' : SYMBOL \pfun VALUE \\
\hline
\# st \leq MaxTableSize \\
\# st' \leq MaxTableSize \\
\hline
\end{array}
$$

Consider the operation that updates the symbol table (in Z, it is conventional to end the names of input variables with a question mark):

$$
\begin{array}{|l}
\hline
Update \\
\Delta SYMBOL\_TABLE \\
sym? : SYMBOL \\
val? : VALUE \\
\hline
st' = st \oplus \{sym? \mapsto val?\} \\
\hline
\end{array}
$$

The pre-defined Z operation $\oplus$ (functional overriding) either adds the pair $(sym?, val?)$ if $sym?$ was not already present, or overrides the previous value with $val?$ if it was.

Schemas can be combined with logical connectives. For example, a full specification for updating the symbol table can be separated into two parts: the successful operation

defined above, and another operation if the table is full. They can then be combined to specify the complete operation:

$$FullUpdate \mathrel{\widehat=} (Update \wedge Success) \vee Overflow$$

Hayes [37, *pp*.51-71] shows how a simple symbol table specification can be used as a component in a more realistic specification of a block-structured symbol table. This ability to construct large specifications from small, understandable components is one of Z's great strengths.

4.2.3 Proof Obligations and Refinement

At the specification level there are various proof obligations that need to be discharged in order to prove that the specification is not vacuous or logically unimplementable. For example, it is necessary to prove the existence of an initial state, and to prove, given a before state and an operation, that an after state can exist.

Z also provides a set of proof obligations for checking the validity of operation refinement and data refinement [74, pp137-141].

4.3 VDM – Vienna Development Method

4.3.1 The Language

VDM has the same mathematical basis as Z, but no schema calculus, making it more difficult to modularize specifications. However, VDM has been around longer than Z, and so the tool support is more mature. The standard texts on VDM are [51,52]. The latter introduces a Logic of Partial Functions to handle VDM proofs, and uses a new VDM syntax.

There are three flavours of VDM specification:

- **implicit :** Jones' pre- and post-condition style, with implicit specification of results via the post-condition.
- **functional :** A function that transforms the input into the output is specified as a combination of simpler functions.
- **procedural :** more akin to a programming language, with do while statements, semi-colons for sequencing, *etc.*

VDM-SL combines these three styles, and is being developed as a BSI standard language [7]. The full language does not yet have a complete set of proof rules, although the implicit style does [52].

VDM uses typed set theory in a subtly different way from Z. Data types are defined using the in-built types (integers, natural numbers, *etc.*), and types may be **refined** by adding a **data type invariant**, constraining their values, producing a **subtype**. VDM provides type constructors allowing structured types to be built from sets, lists (sequences), records and mappings.

Functions are defined by a post-condition in predicate logic which specifies the relationship between the given arguments and the required result. This relationship can be given explicitly, as *result* = *expression*, or implicitly, as a condition.

The "method" of VDM refers to the process of refinement to an implementation. The implicit style has a set of proof rules, and Jones [51] discusses the use of them in rigorous arguments for developing software.

VDM was originally developed by IBM to specify the semantics of the programming language PL/I. It has also been used in a similar way to specify Algol 60 and Pascal [6].

VDM is currently being used to specify the semantics of Modula-2 [49]. It has had to be stretched a little for this purpose. For example, explicit use has been made of lambda-calculus to specify the low-level semantics of co-routines and interrupts. The bulk of the specification uses the classical style to specify the semantics of the language in the usual way, but the implicit style is used to specify the library procedures.

Another VDM use is, for example [82].

4.3.2 EPROS

EPROS (Evolutionary PROtotyping System) [38,39] supports formal specification and rapid prototyping. EPROL (Evolutionary PROtotyping Language) is an executable programming language in which the program can be in the form of (a subset of) VDM. EPROL is expressed in ASCII, with a syntax somewhat different from the usual VDM notation, since EPROL is wider than just the VDM it supports.

One interesting feature of the EPROL approach is its treatment of specifying user interfaces, using an electronic forms approach. [39] discusses this in some detail.

4.4 CCS – Calculus of Communicating Systems

CCS [62,63,79] is an algebra used to describe concurrent systems. More recent work is described in [64], which drops the name CCS in favour of **process calculus**. This is a more general formalism than CCS, in that the latter can be derived as a subsystem of the former.

A system is comprised of a set of interacting **agents**. Each action of an agent is either an interaction with its neighbouring agents, called a **communication**, or it occurs independently of them, in which case it may occur **concurrently** with their actions. These independent actions of an agent can in turn be communications among the components of that agent.

The communication paradigm is of **handshake communication** between agents. A handshake is an indivisible act of communication, experienced simultaneously by both participants. Communication takes place via labelled **ports**. Output port labels are distinguished from input port labels by an overbar, for example: in and $\overline{out}$. A port can be linked to more than one other port. This is a potential source of non-determinism in the system's behaviour.

There are five basic ways of constructing expression for agents:

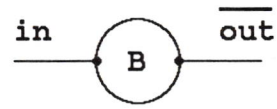
- **prefix :** $\alpha.P$
  Perform action α, then behave like agent P.

- **summation :** $P + Q$
  Behave like P or like Q.
- **composition :** $P \mid Q$
  Behave like P and Q together. P and Q may proceed independently, but may also interact through complementary ports ($port$, $\overline{port}$). Such ports are not hidden, but are still available for further linkage.
- **restriction :** $P\backslash\{l_1, \ldots, l_n\}$
  Internalise (hide) the ports labelled l_i, and also their complements labelled $\overline{l_i}$.
- **relabelling :** $P[l'_1/l_1, \ldots, l'_n/l_n]$
  Relabel the ports l_i as l'_i, and also relabel their complements $\overline{l_i}$ as $\overline{l'_i}$.

As an example, a one place buffer can be defined by:

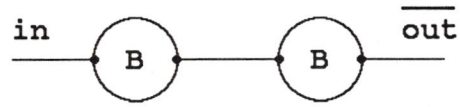
$$B \stackrel{\text{def}}{=} in(x).\overline{out}(x).B$$

This recursively defines the behaviour of the agent B to be an input of x on port in, followed by an output of the same x on port $\overline{out}$, followed by the behaviour of agent B. Its port structure (but not its behaviour) can be represented diagrammatically as



A two place buffer can be defined by taking two one place buffers, B and B. Renaming the out port of the first and the in port of the second as mid, gives $B[mid/out]$ and $B[mid/in]$:

Composing the buffers gives $B[mid/out] \mid B[mid/in]$, and finally hiding the mid port gives the two place buffer $(B[mid/out] \mid B[mid/in])\backslash mid$:



There are various levels of equality between processes, determined by inputs and outputs on their visible ports. One of these is **observation equivalence** between agents.

4.5 CSP – Communicating Sequential Processes

CSP [43] is a language for specifying and reasoning about concurrent systems, modelled as a networks of independent processes that communicate via channels. CSP provides a rich set of operations which can be used to construct new processes from others. Process construction operations include:

- **construction :** $a \rightarrow P$
  a then P. Perform event a, then behave as process P.
- **parallel composition :** $P \parallel Q$
  P in parallel with Q. P and Q can synchronize on a communication.
- **non-deterministic choice :** $P \sqcap Q$
  Perform either process P, or process Q.
- **choice :** $P[]Q$
  Perform either process P, or process Q, depending on events offered by the environment.
- **channel hiding :** $P \backslash c$
  Process P with all communications on channel c made internal to P.
- **interleaving :** $P \mid\mid\mid Q$
- **chaining :** $P >> Q$
  P is chained to Q. The output channel of P is attached to the input channel of Q, and hidden.
- **sequential composition :** $P; Q$
  P followed by Q.
- **conditional choice :** $P \lhd b \rhd Q$
  P if b, else Q.
- **output :** $c!e$
  On channel c output the value of e.
- **input :** $c?x$
  On channel c input to x.

The specifier has to ensure that input communication events and the corresponding output communication events are correctly matched in terms of data type.

A classic problem, often quoted because it is so difficult to solve in a conventional language, is to input cards containing 80 characters each, replace each pair of asterisks with an up-arrow, then output in lines of 125 characters [43, p153]. In CSP, this reduces to defining a separate process for each stage: *UNPACK* inputs sequences of 80 characters, and outputs them one at a time, *SQUASH* replaces the asterisks, and *PACK* inputs single characters, and outputs them 125 at a time. The complete process is simply these chained together:

$$UNPACK >> SQUASH >> PACK$$

Recent work in CSP has involved extending it to include real-time, and to develop methods to allow it to be integrated with state-based specification techniques such as VDM and Z. For a discussion of deadlock freedom in CSP, see [69].

4.6 LOTOS – Language of Temporal Ordering Specification

4.6.1 The Language

LOTOS [48] is a formal language for specifying concurrent or distributed systems. It was initially developed, within ISO (Internation Standards Organisation), for the specification of Open Systems Interconnection (OSI) systems. LOTOS is being adopted as an ISO Draft International Standard.

LOTOS is an algebraic specification technique, developed from the idea that systems can be described by defining the temporal relation between the events in the externally visible behaviour of the system. The formal mathematical model of LOTOS has two components. The first deals with the description of process behaviours and interactions, and is based on CCS. The second component deals with the description of data structures and value expressions. This was originally based on the ADT language ACT ONE [24], but any other well-defined language for the description of data structures can be used.

4.6.2 SEDOS LOTOS Toolset

Esprit project ST410, Software Environment for Design of Open Distributed Systems (SEDOS), has developed a prototype toolset intended to provide machine support for LOTOS users [58].

Any text editor that can output ASCII can be used to produce a text file containing standard LOTOS text. The toolset provides syntax checking and static semantics checking.

HIPPO is the LOTOS simulator. It provides an interactive symbolic execution of a LOTOS specification by stepping through the specification, selecting events from the menu of possible events in each state. Deadlock properties of can be investigated, test sequences can be analysed, and dynamic behaviour can be tested.

4.7 HOL – Higher Order Logic

HOL [28] is a particular notation and machine support system that allows specifications to be written in terms of a higher order logic, which is a logic where predicates can have functions for arguments, unlike the first order logics used in Z and VDM.

HOL has been used in the formal proof of the VIPER microprocessor [17].

4.8 Forest

Forest is the Alvey Formal Requirements Specification project. It includes a methodology to help in the process of extracting formal specifications from informal requirements.

Part of the Forest research has been the development of MAL (Modal Action Logic) as its specification language [50]. This is a many-sorted predicate logic, with three modal components:

- **actions and agents**
  Agents within the system perform actions, which change the state of the system. Actions can be combined to produce more complicated actions by using parallel composition, sequential composition, alternation (choice) and finite iteration.

- **deontic operators**
  The specification describes when actions may, may not, or must occur, by using the three deontic operators **per** (permission), **ref** (refrain) and **obl** (obligation).

- **time**
  Timing requirements and temporal relationships can be specified by using **occ** (occurring), **foa** (first occurrence after), **lob** (last occurrence before), **during**, **cotemp** (simultaneously both true or both false) and **overlaps**.

The temporal aspects of Forest are still under development. Real-time clocks can be specified, but it is a modelling assumption that a clock ticks at a regular rate. The relationship between distinct clocks must be explicitly specified.

4.9 Other Existing Methods

There are many specification languages in addition to the ones detailed above. These include, among others:

- **Affirm :** [68] is an algebraic specification language, used for specification and verification. It is based on rewrite rules, has a consistency and syntax checker.

- **Clear :** [10] is an algebraic specification language that has a complete formal semantics [9,71]. It is a predecessor of OBJ.

- **Estelle :** [8] a formal description technique based on an extended state transition model, used mainly in the communications industry for specifying protocols.

- **Larch :** [30,31] is an algebraic specification language. It has two parts, a *shared language* for writing specifications at a high level, and a range of *interface languages*, each of which provides a bridge to a different style of implementation language.

5 Formal Construction of Programs

5.1 Hoare's Axiomatic Approach

Many of Hoare's papers on axiomatic programming can be found in [44]. Descriptions of the constructive approach to program proving can be found in [22,23,29,42]. Dijkstra's **guarded command** language [21] is formal, and so can be used to construct programs as sketched below, but is close enough to "real" programming languages that the translation to the latter is straightforward.

The notation $\{P\}S\{Q\}$ asserts [1] that if P is true before the execution of program segment S, then Q will be true when S terminates. To say "S is correct" means that the assertion $\{P\}S\{Q\}$ has been proved. P is the **pre-condition**, Q is the **post-condition**.

[1] Hoare originally used the notation was $P\{S\}Q$. The new notation is intended to look more like a program, with P and Q as comments, or annotations.

If $a \Rightarrow b$, then a is said to be **stronger** than b, similarly b is **weaker** than a. This is because b is less restrictive that a; the set of b's states contains all of a's and possibly more.

If it is required to prove (or construct) a program S that satisfies a post-condition $\{R\}$ for a given pre-condition $\{P\}$, it is sufficient to prove that S satisfies $\{R\}$ for the weaker pre-condition Q, where $P \Rightarrow Q$. Hence a valid refinement technique is to **weaken the pre-condition**: which produces a solution that provides all that was required, and more.

$$\frac{P \Rightarrow Q, \quad \{Q\}S\{R\}}{\{P\}S\{R\}}$$

Given a program S and post-condition R, the set of *all* initial states that are transformed by S to states satisfying R themselves satisfy the **weakest pre-condition** on S.

Refinement can also **strengthen the post-condition**:

$$\frac{\{Q\}S\{R\}, \quad R \Rightarrow P}{\{Q\}S\{P\}}$$

Proven correct fragments can then be combined to produce larger program segments using various proof rules. For example, the **Rule of Composition** is:

$$\frac{\{P\}S\{Q\}, \quad \{Q\}T\{R\}}{\{P\}S\,;\,T\{R\}}$$

This states that if P is true before execution of S with Q true after, and if Q is true before T with R true after, then the inference is that P is true before S followed by T with R true after the combined execution.

Programming constructs have properties that can be used when proving program fragments. For example, the **Rule of Alternation** is used to reason about choice:

$$\frac{\{Q \wedge P\}S\{R\}, \quad \{Q \wedge \neg P\}T\{R\}}{\{Q\} \text{ if } P \text{ then } S \text{ else } T\{R\}}$$

and the **Rule of Iteration** is used to reason about loops:

$$\frac{\{P \wedge R \wedge (0 \leq B_0)\}S\{R \wedge (0 \leq B_1 < B_0)\}}{\{R\} \text{ while } P \text{ do } S\{\neg P \wedge R\}}$$

The proof of a loop has two parts. The first is to prove **correctness**, which uses the **loop invariant**, R, a property that is not changed on executing the loop body. R is closely related to the post-condition of the loop. The second part is to prove **termination**, which uses the **loop variant**, B. A variant is a non-negative integer that decreases on every iteration of the loop, but never becomes negative. If such a quantity exists, the loop must terminate, *i.e.* the boolean condition P eventually becomes false.

Correctness is a safety property: *nothing bad can happen*. Termination ios a liveness property: *something eventually happens*. A program that is both correct and terminates is called **totally correct**: *something good always happens*.

As an example of the formal construction of a program snippet, consider the classic specification of the quotient/remainder problem. The requirement is to write a program that, given two inputs x and y, outputs the quotient, $q = x \ div \ y$, and the remainder $r = x \ mod \ y$. This requirement can be specified as a pre- and post-condition:

$$\textbf{pre} : x \geq 0 \wedge y > 0$$
$$\textbf{post} : (q \times y + r = x) \wedge (0 \leq r < y)$$

The following program satisfies this specification (in the interests of clarity, no declaration have been shown; all variables are integers):

$\{\textbf{pre} : x \geq 0 \wedge y > 0\}$
$q := 0;$
$r := x;$
while $r \geq y$ **do**
    $\{\textbf{invariant} : q \times y + r = x\}$
    $\{\textbf{variant} : r\}$
    $q := q + 1;$
    $r := r - y;$
$\{\textbf{post} : (q \times y + r = x) \wedge (0 \leq r < y)\}$

The proof of correctness has two parts, and is an inductive proof. The first step is to prove that the invariant is true just before entering the loop:

$$\begin{aligned} & q_0 \times y + r_0 \\ & \quad = 0 \times y + x \\ & \quad = x \end{aligned}$$

□

The second step is to prove that if the invariant is true after the body of the loop has been executed k times, then it is true after the body has been executed $k + 1$ times:

$$\begin{aligned} & q_{k+1} \times y + r_{k+1} \\ & \quad = (q_k + 1) \times y + (r_k - y) \\ & \quad = q_k \times y + r_k \\ & \quad = x \qquad \text{— by induction hypothesis} \end{aligned}$$

□

Hence, by induction, this proves the first conjunct of the post-condition ($q \times y + r = x$) is satisfied. That part of the second conjunct ($r < y$) is satisfied follows from the negation of the loop boolean. The only thing left to prove is that $0 \leq r$, and that follows from the termination proof.

That the program terminates is given by the loop variant. In the example, the variant is r. It starts off non-negative, since $r = x$ by assignment and $x \geq 0$ from the pre-condition. It is decreased every iteration by an amount $y > 0$. It never becomes negative, because of the loop boolean: $r_k \geq y \Rightarrow r_{k+1} \geq 0$. Hence the loop terminates (and also the last remaining part of the postcondition is satisfied).

So the program fragment is both correct, and terminates. Hence it is **totally correct**.

Much of the skill in this type of reasoning is in constructing the loop invariant and variant. When this style of program proving first started, programs were written, then proved by finding a suitable invariant and variant. This was found to be extremely difficult, often producing ugly, clumsy expressions. The modern approach is to construct the program and proof together. The invariant and variant are constructed from the post-condition, and then used to construct the loop.

5.2 Formal Programming Languages

5.2.1 occam

The programming language occam [46] is derived from CSP [43, pp240-243].

There are laws for reasoning about, and transforming, occam programs [70]. In addition to the conventional laws about sequential programming structures, there are additional ones for the manipulation of parallelism and communications. For example,

```
PAR                    ≡    var := expr
  chan ! expr
  chan ? var
```

Rules such as this allow the transformation of parallel to sequential program fragments, and *vice versa*. This allows a program to be written at a high level of abstraction, then later correctly transformed to a logically equivalent, but possibly more efficient, version. Such formal transformation rules also allow reasoning in order to prove freedom from deadlock, and other desirable properties of the program.

occam allows no implicit type casting, and recognises the fact that, in a computer implementation, arithmetic is not associative. So, in occam, adding three numbers cannot be written as $a + b + c$, but has to be written as either $a + (b + c)$ or $(a + b) + c$.

Although the transputer [47] has been designed to execute occam programs efficiently, occam is a general purpose language that could be executed on any processor. occam itself has been used as a specification language in its own right, although it is probably at too low a level to be a true abstract specification language.

5.2.2 Eiffel

Eiffel [61] is an object-oriented language that has been designed from scratch, rather than having object-oriented extensions bolted on to an existing language [5,19,65,76].

What is interesting about Eiffel from the Formal Methods perspective is the way assertions (pre- and post-conditions, invariants, loop termination) can be included as *part of the program*, by use of the keywords **check**, **require**, **ensure**, **invariant**, and **variant**. As well as helping in the production of correct software, the assertions provide part of the program documentation. Eiffel has an automatic documentation tool, **short**, that extracts information, including the assertions, from the code, to provide up-to-date documentation.

Another part of Eiffel designed in, rather than bolted on, is its disciplined exception handling: "based on a contracting metaphor, under no circumstances should a routine pretend it has succeeded when in fact it has failed to achieve its purpose" [61, p157]. This is achieved by means of the keywords **rescue** and **retry**. Such an approach aids reasoning about program behaviour, and hence aids writing correct programs.

5.2.3 Other Formal Programming Languages

- **Euclid :** Reference [56,57]
- **functional programming languages :** based on lambda calculus.
- **Gypsy :** [1] another language that integrates specification and program into a single text. It supports "entry" and "exit" assertion for blocks, *etc.*
- **Prolog :** [14] based on predicate calculus.

5.3 Code Analysis Tools

As programming languages get stricter and as their compilers get better and do more checking, many common programming faults (for example, use of undeclared variables, use of incompatible types in expressions) can be caught.

There is also a range of tools for checking the code more rigorously. These tools can provide various degrees of checking:

- **control-flow analysis :** for detecting unreachable code, jumping into loops, *etc.*
- **data-flow analysis :** for detecting uninitialized and unused variables, *etc.*
- **semantic analysis :** for checking the code against assertions such as pre- and post-conditions and loop invariants.

One such tool is the Southampton Program Analysis and Development Environment, SPADE [13,15]. This provides a translator from (a 'safe' subset of) the high level language or assemble into SPADE's Functional Description Language (FDL). The various analyses are carried out on the FDL.

Another well known code analysis tool is MALPAS, from RSRE.

6 References

[1] Ambler, A.L., Good, D.I., Burger, W.F., Browne, J.C., Cohen, R.M., Hoch, C.G. & Wells, R.E.: 'Gypsy: a language for specification and implementation of verifiable programs', *SIGPLAN Notices*, 1977, **12**(3), 1-10.

[2] Barringer, H.: 'Up and Down the Temporal Way', *The Computer Journal*, 1987, **30**(2), 134-148.

[3] Barrett, G.: 'Formal Methods Applied to a Floating Point Number System', Monograph PRG-58, (Programming Research Group, Oxford University Computing Laboratory), 1987.

[4] Binmore, K.B.: *Foundations of Analysis Book 1: Logic, Sets and Numbers*, (Cambridge University Press), 1980.

[5] Bobrow, D.G. & Stefik, M.J.: 'LOOPS: an Object Oriented Programming System for Interlisp', (Xerox PARC, USA), 1982.

[6] Bjørner, D. & Jones, C.B.: *Formal Specification and Software Development*, (Prentice Hall), 1982.

[7] BSI IST/5/50: 'VDM Specification Language Protostandard, Context Conditions and Dynamic Semantics', 1989.

[8] Budkowski, S. & Dembinski, P.: 'An Introduction to Estelle: A Specification Language for Distributed Systems', *Computer Networks and ISDN Systems*, 1988, **14**(1), 3-24.

[9] Burstall, R. & Goguen, J.A.: 'The Semantics of Clear, a Specification Language', in Bjørner, D. (Ed.): *Abstract Software Specifications*, Lecture Notes in Computer Science 86, (Springer), 1980, pp 292-332,

[10] Burstall, R. & Goguen, J.A.: 'An Informal Introduction to Specifications using Clear', in Boyer, R.S. & Moore, J.S. (Eds.): *The Correctness Problem in Computer Science*, (Academic Press), 1981, pp 185-213,

[11] Burstall, R., MacQueen, D. & Sannella, D.: 'Hope – An Experimental Applicative Language', Internal Report CSR-62-80, (Edinburgh University), 1980 (updated 1981).

[12] Carrington, D., Duke, D., Duke, R., King, P., Rose, G. & Smith, G.: 'Object-Z: An Object-Oriented Extension to Z', (Department of Computer Science, University of Queensland).

[13] Carré, B.A., Clutterbuck, D.L., Debney, C.W. & O'Neill, I.M.: 'SPADE: the Southampton Program Analysis and Development Environment', in Sommerville, I. (Ed.): *Software Engineering Environments*, (Peter Peregrinus), 1986, 129-134.

[14] Clocksin, W.F. & Mellish, C.S.: *Programming in Prolog*, 3rd Ed., (Springer), 1987.

[15] Clutterbuck, D.L. & Carré, B.A.: 'The verification of low-level code', *Software Engineering Journal*, 1988, May, 97-111.

[16] Cohen, B., Harwood, W.T. & Jackson, M.I.: *The Specification of Complex Systems*, (Addison-Wesley), 1986.

[17] Cohn, A.: 'A proof of correctness of the Viper microprocessor: The first level', Technical Report 104, (University of Cambridge), 1987.

[18] Coleman, D., Gallimore, R.M. & Stavridou, V.: 'The design of a rewrite rule interpreter from algebraic specifications', *IEE Software Engineering Journal*, 1987, July, 95-104.

[19] Cox, B.J.: *Object Oriented Programming – An Evolutionary Approach*, (Addison-Wesley), 1986.

[20] De Millo, R.A., Lipton, R.J. & Perlis, A.J.: 'Social Processes and Proofs of Theorems and Programs', *Comm. ACM*, 1979, **22**(5), 271-280.
see also: 'ACM Forum', *Comm. ACM*, 1979, **22**(11), 621-630.

[21] Dijkstra, E.W.: *A Discipline of Programming*, (Prentice Hall), 1976.

[22] Dijkstra, E.W. & Feijen, W.H.J.: *A Method of Programming*, (Addison-Wesley), 1988.

[23] Dromey, R.G.: *Program Derivation: the development of programs from specifications*, (Addison-Wesley), 1989.

[24] Ehrig, H., Fey, W. & Hansen, H.: 'ACT ONE: An Algebraic Specification Language with two Levels of Semantics', Bericht-Nr. 83-03, (Technical University of Berlin), 1983.

[25] Gallimore, R.M., Coleman, D. & Stavridou, V.: 'UMIST OBJ: a Language for Executable Program Specifications', *The Computer Journal*, 1989, **32**(5), 413-421.

[26] Goguen, J.A. & Meseguer, J.: 'Programming with Parameterised Objects', in Ferrari, D., Bolognane, N. & Goguen, J.A. (Ed.): *Theory and Practice of Software Technology*, (North-Holland), 1983.

[27] Goguen, J.A. & Tardo, J.J.: 'An introduction to OBJ, a Language for Writing and Testing Software Specifications', in *Specification of Reliable Software*, (IEEE), 1979.

[28] Gordon, M.: 'HOL: A machine oriented formulation of higher order logic', Technical Report 68, (University of Cambridge), 1985.

[29] Gries, D.: *The Science of Programming*, (Springer), 1981.

[30] Guttag, J.V., Horning, J.J. & Wing, J.M.: 'Larch in Five Easy Pieces', *Digital Systems Research Center Report*, July, 1985.

[31] Guttag, J.V., Horning, J.J. & Wing, J.M.: 'The Larch Family of Specification Languages', *IEEE Software*, 1985, **2**(5) 24-36.

[32] Halmos, P.R.: *Naive Set Theory*, (Van Nostrand Reinhold), 1960.

[33] Hamilton, A.G.: *Logic for Mathematicians*, (Cambridge University Press), 1978.

[34] Hamilton, A.G.: *Numbers, Sets and Axioms*, (Cambridge University Press), 1982.

[35] Harper, R., Milner, R. & Tofte, M.: 'The Semantics of Standard ML, Version 1', ECS-LFCS-87-36, (Laboratory for the Foundations of Computer Science, Edinburgh University), 1987.

[36] Hayes, I.: 'Specifying the CICS Application Programmer's Interface', Monograph PRG-47, (Programming Research Group, Oxford University Computing Laboratory), 1985.

[37] Hayes, I.: *Specification Case Studies*, (Prentice Hall), 1987.

[38] Hekmatpour, S.: 'EPROS - A Software Engineer's Manual', Technical report No. 86/2, 1986.

[39] Hekmatpour, S. & Ince, D.C.: *Software Prototyping, Formal Methods and VDM*, (Addison-Wesley), 1988.

[40] Henson, M.C.: *Elements of Functional Languages*, (Blackwell Scientific Publications), 1987.

[41] Hindley, J.R. & Seldin, J.P.: *Introduction to Combinators and λ-Calculus*, (Cambridge University Press), 1986.

[42] Hoare, C.A.R.: 'An Axiomatic Approach to Programming', *Comm. ACM*, 1969, **12**(10), 576-580.

[43] Hoare, C.A.R.: *Communicating Sequential Processes*, (Prentice Hall), 1985.

[44] Hoare, C.A.R. & Jones, C.B.: *Essays in Computer Science*, (Prentice Hall), 1989.

[45] Ince, D.C.: *An Introduction to Discrete Mathematics and Formal System Specification*, (Oxford University Press), 1988.

[46] Inmos Ltd.: *occam 2 Reference Manual*, (Prentice Hall), 1988.

[47] Inmos Ltd.: *Transputer Technical Notes*, (Prentice Hall), 1989.

[48] ISO/TC97/SC21: 'LOTOS, a Formal Description Technique based on the Temporal Ordering of Observational Behaviour'. Draft International Standard, ISO/DIS/8807, 1987.

[49] ISO/IEC JTC1 SC22/WG13: '2nd working draft of the Modula 2 Standard', document D103, 1989.

[50] Jeremaes, P., Khosla, S. & Maibaum, T.S.E.: 'A modal (action) logic for requirements specifications', in Brown, P.J. & Barnes, D.J. (Ed.): *Software Engineering '86*, (Peter Peregrinus), 1986.

[51] Jones, C.B.: *Software Development: a rigorous approach*, (Prentice Hall), 1980.

[52] Jones, C.B.: *Systematic Software Development using VDM*, (Prentice Hall), 1986.

[53] King, S., Sørenson, I.H. & Woodcock, J.C.P.: 'Z: Grammar and Concrete and Abstract Syntaxes', Monograph PRG-68, (Programming Research Group, Oxford University Computing Laboratory), 1988.

[54] Knuth, D.E.: *The TEXbook*, (Addison-Wesley), 1984.

[55] Lamport, L.: LATEX: *a Document Preparation System*, (Addison-Wesley), 1986.

[56] Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G. & Popek, G.J.: 'Report on the programming language Euclid', *SIGPLAN Notices*, 1977, **12**(2).

[57] London, R.L., Guttag, J.V., Horning, J.J., Mitchell, B.W. & Popek, G.J.: 'Proof Rules for the Programming Language Euclid', *Acta Informatica*, 1987, **1**, 1-97.

[58] Marshall, A.K.: 'A Prototype Integrated Toolset for LOTOS', SEDOS/C3/15 - Issue 2.1, 1987.

[59] McMorran, M.A. & Nicholls, J.E.: 'Z User Manual', Technical Report TR12.274, (IBM UK Hursley Park), 1989.

[60] Meyer, B.: 'On Formalism in Specifications', *IEEE Software*, 1985, January, 6-26.

[61] Meyer, B.: *Object-oriented Software Construction*, (Prentice Hall), 1988.

[62] Milner, R.: *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, (Springer), 1980.

[63] Milner, R.: 'A Calculus of Communicating Systems', ECS-LFCS-86-7, (Laboratory for the Foundations of Computer Science, Edinburgh University), 1986.

[64] Milner, R.: *Communication and Concurrency*, (Prentice Hall), 1989.

[65] Moon, D.A.: 'Object-Oriented Programming with Flavors', OOPSLA '86, *ACM SIGPLAN Notices*, 1986, **21**(11).

[66] Morgan, C.C. & Sufrin, B.A.: 'Specification of The Unix Filing System', *IEEE Transactions on Software Engineering*, 1982, **10**, 128-142.

[67] Moszkowski, B.: *Executing Temporal Logic Programs*, (Cambridge University Press), 1986.

[68] Musser, D.R.: 'Abstract Data Type Specification in the AFFIRM system', *IEEE Transactions on Software Engineering*, 1980, **SE-6**(1), 24-32.

[69] Roscoe, A.W. & Dathi, N.: 'The Pursuit of Deadlock Freedom', Monograph PRG-57, (Programming Research Group, Oxford University Computing Laboratory), 1986.

[70] Roscoe, A.W. & Hoare, C.A.R.: 'The Laws of occam Programming', Monograph PRG-53, (Programming Research Group, Oxford University Computing Laboratory), 1986.

[71] Sannella, D.T.: 'A New Semantics for Clear', Technical Report CSR-79-81, (Department of Computer Science, Edinburgh University), 1981.

[72] Schuman, S.A. & Pitt, D.H.: 'Object-oriented Subsystem Specification', in Meertens (Ed.): *Program Specification and Transformation*, (North Holland), 1987.

[73] Spivey, J.M.: *Understanding Z: a specification language and its formal semantics*, (Cambridge University Press), 1988.

[74] Spivey, J.M.: *The Z Notation: a reference manual*, (Prentice Hall), 1989.

[75] Stepney, S. & Lord, S.P.: 'Formal Specification of an Access Control System', *Software – Practice and Experience*, 1987, **17**(9), 575-593.

[76] Stroustrup, B.: *The C++ Programming Language*, (Addison-Wesley), 1986.

[77] Turner, D.: 'An Overview of Miranda', *ACM SIGPLAN Notices*, 1986, **21**(12), 158-166.

[78] Turner, K.J.: *Formal Description Techniques*, Forte 88 Proceedings, (North Holland), 1989.

[79] Walker, D.: 'Introduction to a Calculus of Communicating Systems', ECS-LFCS-87-22, (Laboratory for the Foundations of Computer Science, Edinburgh University), 1987.

[80] Wikström, Å.: *Functional Programming using Standard ML*, (Prentice Hall), 1987.

[81] Woodcock, J.C.P.: 'Structuring Specifications in Z', *Software Engineering Journal*, 1989, January, pp. 51-66.

[82] Woodcock, J.C.P. & Dickson, B.: 'Using VDM with Rely and Guarantee Conditions: Experiences from a Real Project', *Proc. VDM Symposium 88*, Lecture Notes in Computer Science 328, (Springer) 1988.

[83] Woodcock, J.C.P. & Loomes, M.: *Software Engineering Mathematics*, (Pitman) 1988.