Formal Methods

18

Key Topics

Vienna Development Method
Z Specification Language
B Method
Process Calculi
Model-oriented approach
Axiomatic approach
Usability of Formal Methods

18.1 Introduction

The term 'formal methods' refers to various mathematical techniques used for the formal specification and development of software. They consist of a formal specification language, and employ a collection of tools to support the syntax checking of the specification, as well as the proof of properties of the specification. They allow questions to be asked about what the system does independently of the implementation.

The use of mathematical notation avoids speculation about the meaning of phrases in an imprecisely worded natural language description of a system. Natural language is inherently ambiguous, whereas mathematics employs a precise rigorous notation. Spivey [1] defines formal specification as:

G. O'Regan, *Guide to Discrete Mathematics*, Texts in Computer Science,
DOI 10.1007/978-3-319-44561-8_18

Definition 18.1 (*Formal Specification*) Formal specification is the use of mathematical notation to describe in a precise way the properties that an information system must have, without unduly constraining the way in which these properties are achieved.

The formal specification thus becomes the key reference point for the different parties involved in the construction of the system. It may be used as the reference point in the requirements; program implementation; testing and program documentation. It promotes a common understanding for all those concerned with the system. The term '*formal methods*' is used to describe a formal specification language and a method for the design and implementation of computer systems.

The specification is written in a mathematical language, and the implementation is derived from the specification via step-wise refinement.[1] The refinement step makes the specification more concrete and closer to the actual implementation. There is an associated proof obligation to demonstrate that the refinement is valid, and that the concrete state preserves the properties of the more abstract state. Thus, assuming that the original specification is correct, and the proofs of correctness of each refinement step are valid, then there is a very high degree of confidence in the correctness of the implemented software.

Step-wise refinement is illustrated as follows: the initial specification S is the initial model M_0; it is then refined into the more concrete model M_1, and M_1 is then refined into M_2, and so on until the eventual implementation $M_n = E$ is produced.

$$S = M_0 \subseteq M_1 \subseteq M_2 \subseteq M_3 \subseteq \cdots \subseteq M_n = E$$

Requirements are the foundation of the system to be built, and irrespective of the best design and development practices, the product will be incorrect if the requirements are incorrect. The objective of *requirements validation* is to ensure that the requirements reflect what is actually required by the customer (*in order to build the right system*). Formal methods may be employed to model the requirements, and the model exploration yields further desirable or undesirable properties. The ability to prove that certain properties are true of the specification is very valuable, especially in safety critical and security critical applications. These properties are logical consequences of the definition of the requirements, and, where appropriate, the requirements may need to be amended. Thus, formal methods may be employed in a sense to debug the requirements during requirements validation.

The use of formal methods generally leads to more robust software and to increased confidence in its correctness. The challenges involved in the deployment

[1] It is debatable whether step-wise refinement is cost effective in mainstream software engineering, as it involves re-writing a specification ad nauseam. It is time-consuming, as significant time is required to prove that each refinement step is valid.

Table 18.1 Criticisms of formal methods

No.	Criticism
1	Often the formal specification is as difficult to read as the program[a]
2	Many formal specifications are wrong[b]
3	Formal methods are strong on syntax but provide little assistance in deciding on what technical information should be recorded using the syntax[c]
4	Formal specifications provide a model of the proposed system. However, a precise unambiguous mathematical statement of the requirements is what is needed[d]
5	Step-wise refinement is unrealistic.[e] It is like, for example, deriving a bridge from the description of a river and the expected traffic on the bridge. There is always a need for a creative step in design
6	Much unnecessary mathematical formalisms have been developed rather than using the available classical mathematics[f]

[a]Of course, others might reply by saying that some of Parnas's tables are not exactly intuitive, and that the notation he employs in some of his tables is quite unfriendly. The usability of all of the mathematical approaches needs to be enhanced if they are to be taken seriously by industrialists
[b]Obviously, the formal specification must be analysed using mathematical reasoning and tools to provide confidence in its correctness. The validation may be carried out using mathematical proof of key properties of the specification; software inspections; or specification animation
[c]VDM includes a method for software development as well as the specification language
[d]Models are extremely valuable as they allow simplification of the reality. A mathematical study of the model demonstrates whether it is a suitable representation of the system. Models allow properties of the proposed requirements to be studied prior to implementation
[e]Step-wise refinement involves rewriting a specification with each refinement step producing a more concrete specification (that includes code and formal specification) until eventually the detailed code is produced. However, tool support may make refinement easier
[f]Approaches such as VDM or Z are useful in that they add greater rigour to the software development process. Classical mathematics is familiar to students and therefore it is desirable that new formalisms are introduced only where absolutely necessary

of formal methods in an organization include the education of staff in formal specification, as the use of these mathematical techniques may be a culture shock to many staff.

Formal methods have been applied to a diverse range of applications, including the security critical field; the safety critical field; the railway sector; microprocessor verification; the specification of standards, and the specification and verification of programs.

Parnas and others have criticized formal methods on the following grounds (Table 18.1).

However, formal methods are potentially quite useful and reasonably easy to use. The use of a formal method such as Z or VDM forces the software engineer to be precise and helps to avoid ambiguities present in natural language. Clearly, a formal specification should be subject to a peer review to provide confidence in its correctness. New formalisms need to be intuitive to be usable by practitioners. The advantage of classical mathematics is that it is familiar to students.

18.2 Why Should We Use Formal Methods?

There is a strong motivation to use best practice in software engineering in order to produce software adhering to high-quality standards. Quality problems with software may cause minor irritations or major damage to a customer's business including loss of life.[2] Formal methods are a leading-edge technology that may help companies to reduce the occurrence of defects in software products. Brown [2] argues that for the safety critical field that:

Comment 18.1 (Missile Safety) *Missile systems must be presumed dangerous until shown to be safe, and that the absence of evidence for the existence of dangerous errors does not amount to evidence for the absence of danger.*

This suggests that companies in the safety critical field need to demonstrate that every reasonable practice was taken to prevent the occurrence of defects. One such practice is the use of formal methods, and its exclusion may need to be justified in some domains. It is quite possible that a software company may be sued for software which injures a third party,[3] and this suggests that companies will need a rigorous quality assurance system to prevent the occurrence of defects.

There is some evidence to suggest that the use of formal methods provides savings in the cost of the project. For example, a 9 % cost saving is attributed to the use of formal methods during the CICS project; the T800 project attributes a 12-month reduction in testing time to the use of formal methods. These are discussed in more detail in Chap. 1 of [3].

The use of formal methods is mandatory in certain circumstances. The Ministry of Defence in the United Kingdom issued two safety-critical standards[4] in the early 1990s related to the use of formal methods in the software development lifecycle.

The first is Defence Standard 00-55, "The Procurement of safety critical software in defense equipment" [4] which makes it mandatory to employ formal methods in safety-critical software development in the UK; and mandates the use of formal proof that the most crucial programs correctly implement their specifications.

The second is Def Stan 00-56 "Hazard analysis and safety classification of the computer and programmable electronic system elements of defense equipment" [5]. The objective of this standard is to provide guidance to identify which systems or parts of systems being developed are safety-critical and thereby require the use of formal methods. This proposed system is subject to an initial hazard analysis to determine whether there are safety-critical parts.

[2]We mentioned the serious problems with the Therac-25 radiotherapy machine in Chap. 17.

[3]A comprehensive disclaimer of responsibility for problems (rather than a guarantee of quality) accompany most software products, and so the legal aspects of licensing software may protect software companies from litigation. However, greater legal protection for the customer can be built into the contract between the supplier and the customer for bespoke-software development.

[4]The U.K. Defence Standards 0055 and 0056 have been revised in recent years to be less prescriptive on the use of formal methods.

The reaction to these defence standards 00-55 and 00-56 was quite hostile initially, as most suppliers were unlikely to meet the technical and organization requirements of the standard [6]. The standards were subsequently revised to be less prescriptive on the use of formal methods.

18.3 Applications of Formal Methods

Formal methods have been employed to verify correctness in the nuclear power industry, the aerospace industry, the security technology area, and the railroad domain. These sectors are subject to stringent regulatory controls to ensure safety and security. Several organizations have piloted formal methods with varying degrees of success. These include IBM, who developed VDM at its laboratory in Vienna; IBM (Hursley) piloted the *Z* formal specification language on the CICS (Customer Information Control System) project.

The mathematical techniques developed by Parnas (i.e., tabular expressions) have been employed to specify the requirements of the A-7 aircraft as part of a research project for the US Navy.[5] Tabular expressions have also been employed for the software inspection of the automated shutdown software of the Darlington Nuclear power plant in Canada.[6] These are two successful uses of mathematical techniques in software engineering.

There are examples of the use of formal methods in the railway domain, and examples dealing with the modeling and verification of a railroad gate controller and railway signaling are described in [3]. Clearly, it is essential to verify safety critical properties such as "when the train goes through the level crossing then the gate is closed".

18.4 Tools for Formal Methods

A key criticism of formal methods is the limited availability of tools to support the software engineer in writing a formal specification or in conducting proof. Many of the early tools were criticized as not being of industrial strength. However, in recent years more advanced tools to support the software engineer's work in formal specification and formal proof have become available, and this should continue in the coming years.

[5]However, the resulting software was never actually deployed on the A-7 aircraft.

[6]This was an impressive use of mathematical techniques and it has been acknowledged that formal methods must play an important role in future developments at Darlington. However, given the time and cost involved in the software inspection of the shutdown software some managers have less enthusiasm in shifting from hardware to software controllers [7].

The tools include syntax checkers that determine whether the specification is syntactically correct; specialized editors which ensure that the written specification is syntactically correct; tools to support refinement; automated code generators that generate a high-level language corresponding to the specification; theorem provers to demonstrate the presence or absence of key properties and to prove the correctness of refinement steps, and to identify and resolve proof obligations; and specification animation tools where the execution of the specification can be simulated.

The *B*-Toolkit from *B*-Core is an integrated set of tools that supports the *B*-Method. These include syntax and type checking, specification animation, proof obligation generator, an auto-prover, a proof assistor, and code generation. This allows, in theory, a complete formal development from initial specification to final implementation to be achieved, with every proof obligation justified, leading to a provably correct program.

The IFAD Toolbox[7] is a support tool for the VDM-SL specification language, and it includes support for syntax and type checking, an interpreter and debugger to execute and debug the specification, and a code generator to convert from VDM-SL to C++. It also includes support for graphical notations such as the OMT/UML design notations.

18.5 Approaches to Formal Methods

There are two key approaches to formal methods: namely the model-oriented approach of VDM or *Z,* and the algebraic or axiomatic approach of the process calculi such as the calculus communicating systems (CCS) or communicating sequential processes (CSP).

18.5.1 Model-Oriented Approach

The *model-oriented* approach to specification is based on mathematical models, and a model is a mathematical representation or abstraction of a physical entity or system. The model aims to provide a mathematical explanation of the behaviour of the physical world, and it is considered suitable if its properties closely match those of the system being modeled. A model will allow predictions of future behaviour to be made, and many models are employed in the physical world (e.g., weather forecasting system).

It is fundamental to explore the model to determine its *adequacy*, and to determine the extent to which it explains the underlying physical behaviour, and allows predictions of future behaviour to be made. This will determine its

[7]The IFAD Toolbox has been renamed to VDMTools as IFAD sold the VDM Tools to CSK in Japan.

acceptability as a representation of the physical world. Models that are ineffective will be replaced with models that offer a better explanation of the manifested physical behaviour. There are many examples in science of the replacement of one theory by a newer one. For example, the Copernican model of the universe replaced the older Ptolemaic model, and Newtonian physics was replaced by Einstein's theories on relativity [8].

The model-oriented approach to software development involves defining an abstract model of the proposed software system. The model acts as a representation of the proposed system, and the model is then explored to assess its suitability. The exploration of the model takes the form of model interrogation, i.e., asking questions and determining the effectiveness of the model in answering the questions. The modeling in formal methods is typically performed via elementary discrete mathematics, including set theory, sequences, functions and relations.

VDM and *Z* are model-oriented approaches to formal methods. VDM arose from work done in the IBM laboratory in Vienna in formalizing the semantics for the PL/1 compiler, and it was later applied to the specification of software systems. The origin of the *Z* specification language is in work done at Oxford University in the early 1980s.

18.5.2 Axiomatic Approach

The *axiomatic approach* focuses on the properties that the proposed system is to satisfy, and there is no intention to produce an abstract model of the system. The required properties and behaviour of the system are stated in mathematical notation. The difference between the axiomatic specification and a model-based approach is may be seen in the example of a stack.

The stack includes operators for pushing an element onto the stack and popping an element from the stack. The properties of *pop* and *push* are explicitly defined in the axiomatic approach. The model-oriented approach constructs an explicit model of the stack and the operations are defined in terms of the effect that they have on the model. The specification of the *pop* operation on a stack is given by axiomatic properties, for example, $pop(push(s, x)) = s$.

Comment 18.2 (Axiomatic Approach) *The property-oriented approach has the advantage that the implementer is not constrained to a particular choice of implementation, and the only constraint is that the implementation must satisfy the stipulated properties.*

The emphasis is on the required properties of the system, and implementation issues are avoided. The focus is on the specification of the underlying behaviour, and properties are typically stated using mathematical logic or higher-order logics. Mechanized theorem-proving techniques may be employed to prove results.

One potential problem with the axiomatic approach is that the properties specified may not be satisfiable in any implementation. Thus, whenever a 'formal

axiomatic theory' is developed a corresponding 'model' of the theory must be identified, in order to ensure that the properties may be realized in practice. *That is, when proposing a system that is to satisfy some set of properties, there is a need to prove that there is at least one system that will satisfy the set of properties*.

18.6 Proof and Formal Methods

A mathematical proof typically includes natural language and mathematical symbols, and often many of the tedious details of the proof are omitted. The proof of a conjecture may be by a 'divide and conquer' technique; i.e., breaking the conjecture down into subgoals and then attempting to prove the subgoals. Many proofs in formal methods are concerned with crosschecking the details of the specification, or checking the validity of refinement steps, or checking that certain properties are satisfied by the specification. There are often many tedious lemmas to be proved, and theorem provers[8] are essential in assisting with this. Machine proof needs to be explicit, and reliance on some brilliant insight is avoided. Proofs by hand are notorious for containing errors or jumps in reasoning, while machine proofs are explicit but are often extremely lengthy and unreadable (e.g., the actual machine proof of correctness of the VIPER microprocessor[9] [6] consisted of several million formulae).

A formal mathematical proof consists of a sequence of formulae, where each element is either an axiom or derived from a previous element in the series by applying a fixed set of mechanical rules.

Theorem provers are invaluable in resolving many of the thousands of proof obligations that arise from a formal specification, and it is not feasible to apply formal methods in an industrial environment without the use of machine-assisted proof. Automated theorem proving is difficult, as often mathematicians prove a theorem with an initial intuitive feeling that the theorem is true. Human intervention to provide guidance or intuition improves the effectiveness of the theorem prover.

The proof of various properties about a program increases confidence in its correctness. However, an absolute proof of correctness[10] is unlikely except for the most trivial of programs. A program may consist of legacy software that is assumed to work; a compiler that is assumed to work correctly creates it. Theorem provers

[8]Many existing theorem provers are difficult to use and are for specialist use only. There is a need to improve their usability.

[9]This verification was controversial with RSRE and Charter overselling VIPER as a chip design that conforms to its formal specification.

[10]This position is controversial with others arguing that if correctness is defined mathematically then the mathematical definition (i.e. formal specification) is a theorem, and the task is to prove that the program satisfies the theorem. They argue that the proofs for non-trivial programs exist, and that the reason why there are not many examples of such proofs is due to a lack of mathematical specifications.

are programs that are assumed to function correctly. The best that formal methods can claim is increased confidence in correctness of the software, rather than an absolute proof of correctness.

18.7 The Future of Formal Methods

The debate concerning the level of use of mathematics in software engineering is still ongoing. Most practitioners are against the use of mathematics and avoid its use. They tend to employ methodologies such as software inspections and testing to improve confidence in the correctness of the software. Industrialists often need to balance conflicting needs such as quality; cost; and aggressive time pressures. They argue that commercial realities dictate that appropriate methodologies and techniques are required that allow them to achieve their business goals in a timely manner.

The other camp argues that the use of mathematics is essential in the delivery of high-quality and reliable software, and that if a company does not place sufficient emphasis on quality it will pay the price in terms of a poor reputation in the market place.

It is generally accepted that mathematics and formal methods must play a role in the safety critical and security critical fields. Apart from that the extent of the use of mathematics is a hotly disputed topic. The pace of change in the world is extraordinary, and companies face major competitive pressures in a global market place. It is unrealistic to expect companies to deploy formal methods unless they have clear evidence that it will support them in delivering commercial products to the market place ahead of their competition, at the right price and with the right quality. Formal methods need to prove that it can do this if it wishes to be taken seriously in mainstream software engineering. The issue of technology transfer of formal methods to industry is discussed in [9].

18.8 The Vienna Development Method

VDM dates from work done by the IBM research laboratory in Vienna. This group was specifying the semantics of the PL/1 programming language using an operational semantic approach (discussed in Chap. 12). That is, the semantics of the language were defined in terms of a hypothetical machine, which interprets the programs of that language [10, 11]. Later work led to the Vienna Development Method (VDM) with its specification language, Meta IV. This was used to give the denotational semantics of programming languages; i.e., a mathematical object (set, function, etc.) is associated with each phrase of the language [11]. The mathematical object is termed the *denotation* of the phrase.

VDM is a *model-oriented approach* and this means that an explicit model of the state of an abstract machine is given, and operations are defined in terms of this state. Operations may act on the system state, taking inputs, and producing outputs as well as a new system state. Operations are defined in a precondition and post-condition style. Each operation has an associated proof obligation to ensure that if the precondition is true, then the operation preserves the system invariant. The initial state itself is, of course, required to satisfy the system invariant.

VDM uses keywords to distinguish different parts of the specification, e.g., preconditions, postconditions, as introduced by the keywords *pre* and *post,* respectively. In keeping with the philosophy that formal methods specifies *what* a system does as distinct from *how*, VDM employs post-conditions to stipulate the effect of the operation on the state. The previous state is then distinguished by employing *hooked variables*, e.g., $\overline{v}$, and the postcondition specifies the new state which is defined by a logical predicate relating the pre-state to the post-state.

VDM is more than its specification language VDM-SL, and is, in fact, a software development method, with rules to verify the steps of development. The rules enable the executable specification, i.e., the detailed code, to be obtained from the initial specification via refinement steps. Thus, we have a sequence $S = S_0, S_1, \ldots, S_n = E$ of specifications, where S is the initial specification, and E is the final (executable) specification.

Retrieval functions enable a return from a more concrete specification to the more abstract specification. The initial specification consists of an initial state, a system state, and a set of operations. The system state is a particular domain, where a domain is built out of primitive domains such as the set of natural numbers, etc., or constructed from primitive domains using domain constructors such as Cartesian product, disjoint union, etc. A domain-invariant predicate may further constrain the domain, and a *type* in VDM reflects a domain obtained in this way. Thus, a type in VDM is more specific than the signature of the type, and thus represents values in the domain defined by the signature, which satisfy the domain invariant. In view of this approach to types, it is clear that VDM types may not be 'statically type checked'.

VDM specifications are structured into modules, with a module containing the module name, parameters, types, operations, etc. Partial functions occur frequently in computer science as many functions, may be undefined, or fail to terminate for some arguments in their domain. VDM addresses partial functions by employing nonstandard logical operators, namely the logic of partial functions (LPFs), which was discussed in Chap. 16.

VDM has been used in industrial projects, and its tool support includes the IFAD Toolbox.[11] There are several variants of VDM, including VDM^{++}, the object-oriented extension of VDM, and the Irish school of the VDM, which is discussed in the next section.

[11]The VDM Tools are now available from the CSK Group in Japan.

18.9 VDM♣, the Irish School of VDM

The Irish School of VDM is a variant of standard VDM, and is characterized by [12] its constructive approach, classical mathematical style, and its terse notation. This method aims to combine the *what* and *how* of formal methods in that its terse specification style stipulates in concise form *what* the system should do; furthermore, the fact that its specifications are constructive (or functional) means that the *how* is included with the *what*. However, it is important to qualify this by stating that the how as presented by VDM♣ is not directly executable, as several of its mathematical data types have no corresponding structure in high-level programming languages or functional languages. Thus, a conversion or reification of the specification into a functional or higher level language must take place to ensure a successful execution. Further, the fact that a specification is constructive is no guarantee that it is a good implementation strategy, if the construction itself is naive.

The Irish school follows a similar development methodology as in standard VDM, and is a model-oriented approach. The initial specification is presented, with initial state and operations defined. The operations are presented with preconditions; however, no postcondition is necessary as the operation is 'functionally' (i.e., explicitly) constructed.

There are proof obligations to demonstrate that the operations preserve the invariant. That is, if the precondition for the operation is true, and the operation is performed, then the system invariant remains true after the operation. The philosophy is to exhibit existence *constructively* rather than a theoretical proof of existence that demonstrates the existence of a solution without presenting an algorithm to construct the solution.

The school avoids the existential quantifier of predicate calculus and reliance on logic in proof is kept to a minimum, and emphasis instead is placed on equational reasoning. Structures with nice algebraic properties are sought, and one nice algebraic structure employed is the monoid, which has closure, associativity, and a unit element. The concept of isomorphism is powerful, reflecting that two structures are essentially identical, and thus we may choose to work with either, depending on which is more convenient for the task in hand.

The school has been influenced by the work of Polya and Lakatos. The former [13] advocated a style of problem solving characterized by first considering an easier sub-problem, and considering several examples. This generally leads to a clearer insight into solving the main problem. Lakatos's approach to mathematical discovery [14] is characterized by heuristic methods. A primitive conjecture is proposed and if global counter-examples to the statement of the conjecture are discovered, then the corresponding *hidden lemma* for which this global counterexample is a local counter example is identified and added to the statement of the primitive conjecture. The process repeats, until no more global counterexamples are found. A skeptical view of absolute truth or certainty is inherent in this.

Partial functions are the norm in VDM♣, and as in standard VDM, the problem is that functions may be undefined, or fail to terminate for several of the arguments in their domain. The logic of partial functions (LPFs) is avoided, and instead care is taken with recursive definitions to ensure termination is achieved for each argument. Academic and industrial projects have been conducted using the method of the Irish school, but at this stage tool support is limited.

18.10 The *Z* Specification Language

Z is a formal specification language founded on Zermelo set theory, and it was developed by Abrial at Oxford University in the early 1980s. It is a model-oriented approach where an explicit model of the state of an abstract machine is given, and the operations are defined in terms of the effect on the state. It includes a mathematical notation that is similar to VDM, and it employs the visually striking schema calculus, which consists essentially of boxes, with these boxes or schemas used to describe operations and states. The schema calculus enables schemas to be used as building blocks and combined with other schemas. The *Z* specification language was published as an ISO standard (ISO/IEC 13568:2002) in 2002.

The schema calculus is a powerful means of decomposing a specification into smaller pieces or schemas. This helps to make *Z* specification highly readable, as each individual schema is small in size and self-contained. The exception handling is done by defining schemas for the exception cases, and these are then combined with the original operation schema. Mathematical data types are used to model the data in a system and these data types obey mathematical laws. These laws enable simplification of expressions and are useful with proofs.

Operations are defined in a precondition/postcondition style. However, the precondition is implicitly defined within the operation; i.e., it is not separated out as in standard VDM. Each operation has an associated proof obligation to ensure that if the precondition is true, then the operation preserves the system invariant. The initial state itself is, of course, required to satisfy the system invariant. Postconditions employ a logical predicate which relates the pre-state to the post-state, and the post-state of a variable v is given by priming, e.g., v'. Various conventions are employed, e.g., *v?* indicates that v is an input variable and *v!* indicates that v is an output variable. The symbol Ξ *Op* operation indicates that this operation does not affect the state, whereas Δ *Op* indicates that this operation that affects the state.

Many data types employed in *Z* have no counterpart in standard programming languages. It is therefore important to identify and describe the concrete data structures that will ultimately represent the abstract mathematical structures. The operations on the abstract data structures may need to be refined to yield operations on the concrete data structure that yield equivalent results. For simple systems, direct refinement (i.e., one step from abstract specification to implementation) may be possible; in more complex systems, deferred refinement is employed, where a sequence of increasingly concrete specifications are produced to yield the

executable specification eventually.*Z* has been successfully applied in industry, and one of its well-known successes is the CICS project at IBM Hursley in England. *Z* is described in more detail in Chap. 19.

18.11 The *B* Method

The *B-Technologies* [15] consist of three components: a method for software development, namely the *B*-Method; a supporting set of tools, namely, the *B*-Toolkit; and a generic program for symbol manipulation, namely, the *B*-Tool (from which the *B*-Toolkit is derived). The *B*-Method is a model-oriented approach and is closely related to the *Z* specification language. Abrial developed the *B* specification language, and every construct in the language has a set theoretic counterpart, and the method is founded on Zermelo set theory. Each operation has an explicit precondition.

One key purpose [15] of the *abstract machine* in the *B*-Method is to provide encapsulation of variables representing the state of the machine and operations that manipulate the state. Machines may refer to other machines, and a machine may be introduced as a refinement of another machine. The abstract machines are specification machines, refinement machines, or implementable machines. The *B*-Method adopts a layered approach to design where the design is gradually made more concrete by a sequence of design layers. Each design layer is a refinement that involves a more detailed implementation in terms of abstract machines of the previous layer. The design refinement ends when the final layer is implemented purely in terms of library machines. Any refinement of a machine by another has associated proof obligations, and proof is required to verify the validity of the refinement step.

Specification animation of the Abstract Machine Notation (AMN) specification is possible with the *B*-Toolkit, and this enables typical usage scenarios of the AMN specification to be explored for requirements validation. This is, in effect, an early form of testing, and it may be used to demonstrate the presence or absence of desirable or undesirable behavior. Verification takes the form of a proof to demonstrate that the invariant is preserved when the operation is executed within its precondition, and this is performed on the AMN specification with the *B*-Toolkit.

The *B*-Toolkit provides several tools that support the *B*-Method, and these include syntax and type checking; specification animation, proof obligation generator, auto prover, proof assistor, and code generation. Thus, in theory, a complete formal development from initial specification to final implementation may be achieved, with every proof obligation justified, leading to a provably correct program.

The *B*-Method and toolkit have been successfully applied in industrial applications, including the CICS project at IBM Hursley in the United Kingdom. The automated support provided has been cited as a major benefit of the application of the *B*-Method and the *B*-Toolkit.

18.12 Predicate Transformers and Weakest Preconditions

The precondition of a program S is a predicate, i.e., a statement that may be true or false, and it is usually required to prove that if the precondition Q is true,; i.e., $\{Q\}$ S $\{R\}$, then execution of S is guaranteed to terminate in a finite amount of time in a state satisfying R.

The weakest precondition of a command S with respect to a postcondition R represents the set of all states such that if execution begins in any one of these states, then execution will terminate in a finite amount of time in a state with R true [16]. These set of states may be represented by a predicate Q', so that $wp(S, R) = wp_S(R) = Q'$, and so wp_S is a predicate transformer, i.e., it may be regarded as a function on predicates. The weakest precondition is the precondition that places the fewest constraints on the state than all of the other preconditions of (S,R). That is, all of the other preconditions are stronger than the weakest precondition.

The notation $Q\{S\}R$ is used to denote partial correctness and indicates that if execution of S commences in any state satisfying Q, and if execution terminates, then the final state will satisfy R. Often, a predicate Q which is stronger than the weakest precondition $wp(S,R)$ is employed, especially where the calculation of the weakest precondition is nontrivial. Thus, a stronger predicate Q such that $Q \Rightarrow wp$ (S, R) is sometimes employed.

There are many properties associated with the weakest preconditions, and these may be used to simplify expressions involving weakest preconditions, and in determining the weakest preconditions of various program commands such as assignments, iterations, etc. Weakest preconditions may be used in developing a proof of correctness of a program in parallel with its development [17].

An imperative program may be regarded as a predicate transformer. This is since a predicate P characterizes the set of states in which the predicate P is true, and an imperative program may be regarded as a binary relation on states, which may be extended to a function F, leading to the Hoare triple $P\{F\}Q$. That is, the program F acts as a predicate transformer with the predicate P regarded as an input assertion, i.e., a Boolean expression that must be true before the program F is executed, and the predicate Q is the output assertion, which is true if the program F terminates (where F commenced in a state satisfying P).

18.13 The Process Calculi

The objectives of the process calculi [18] are to provide mathematical models that provide insight into the diverse issues involved in the specification, design, and implementation of computer systems which continuously act and interact with their environment. These systems may be decomposed into sub-systems that interact with each other and their environment.

The basic building block is the *process*, which is a mathematical abstraction of the interactions between a system and its environment. A process that lasts indefinitely may be specified recursively. Processes may be assembled into systems; they may execute concurrently; or communicate with each other. Process communication may be synchronized, and this takes the form of a process outputting a message simultaneously to another process inputting a message. Resources may be shared among several processes. Process calculi such as CSP [18] and CCS [19] have been developed to enrich the understanding of communication and concurrency, and these calculi obey a rich collection of mathematical laws.

The expression $(a \,?\, P)$ in CSP describes a process which first engages in event a, and then behaves as process P. A recursive definition is written as $(\mu X) \cdot \mathrm{F}(X)$, and the example of a simple chocolate vending machine is given recursively as:

$$\mathrm{VMS} = \mu X : \{coin, choc\} \cdot (coin \,?\, (choc \,?\, X))$$

The simple vending machine has an alphabet of two symbols, namely, *coin* and *choc*. The behaviour of the machine is that a coin is entered into the machine, and then a chocolate selected and provided.

CSP processes use channels to communicate values with their environment, and input on channel c is denoted by $(c?.x\ P_x)$. This describes a process that accepts any value x on channel c, and then behaves as process P_x. In contrast, $(c!e\ P)$ defines a process which outputs the expression e on channel c and then behaves as process P.

The π-calculus is a process calculus based on names. Communication between processes takes place between known channels, and the name of a channel may be passed over a channel. There is no distinction between channel names and data values in the π-calculus. The output of a value v on channel a is given by $\bar{a}v$; i.e., output is a negative prefix. Input on a channel a is given by $a(x)$, and is a positive prefix. Private links or restrictions are given by $(x)P$ in the π-calculus.

18.14 The Parnas Way

Parnas has been influential in the computing field, and his ideas on the specification, design, implementation, maintenance, and documentation of computer software remain important. He advocates a solid classical engineering approach to developing software, and he argues that the role of an engineer is to apply scientific principles and mathematics in designing and developing software products. His main contributions to software engineering are summarized in Table 18.2.

Table 18.2 Parnas's contributions to software engineering

Area	Description
Tabular expressions	These are mathematical tables for specifying requirements, and enable complex predicate logic expressions to be represented in a simpler form
Mathematical documentation	He advocates the use of precise mathematical documentation
Requirements specification	He advocates the use of mathematical relations to specify the requirements precisely
Software design	He developed information hiding which is used in object-oriented design[a], and allows software to be designed for change [21]. Every information-hiding module has an interface that provides the only means to access the services provided by the modules. The interface hides the module's implementation
Software inspections	His approach requires the reviewers to take an active part in the inspection. They are provided with a list of questions by the author and their analysis involves the production of mathematical table to justify the answers
Predicate logic	He developed an extension of the predicate calculus to deal with partial functions. This approach preserves the classical two-valued logic and deals with undefined values that may occur in predicate logic expressions

[a]It is surprising that many in the object-oriented world seem unaware that information hiding goes back to the early 1970s and many have never heard of Parnas

18.15 Usability of Formal Methods

There are practical difficulties associated with the use of formal methods. It seems to be assumed that programmers and customers are willing to become familiar with the mathematics used in formal methods. There is little evidence to suggest that customers in mainstream organizations would be prepared to use formal methods.[12] Customers are concerned with their own domain and speak the technical language of that domain.[13] Often, the use of mathematics is an alien activity that bears little resemblance to their normal work. Programmers are interested in programming rather than in mathematics, and generally have no interest in becoming mathematicians.[14]

[12]The domain in which the software is being used will influence the willingness or otherwise of the customers to become familiar with the mathematics required. There is very little interest from customers in mainstream software engineering, and the perception is that formal methods are difficult to use. However, in some domains such as the regulated sector there is a greater willingness of customers to become familiar with the mathematical notation.

[13]The author's experience is that most customers have a very limited interest in using mathematics.

[14]Mathematics that is potentially useful to software engineers was discussed in Chap. 17.

Table 18.3 Techniques for validation of formal specification

Technique	Description
Proof	This involves demonstrating that the formal specification adheres to key properties of the requirements. The implementation will need to preserve these properties also
Software inspections	This involves a Fagan like inspection to perform the validation. It may involve comparing an informal set of requirements (unless the customer has learned the formal method) with the formal specification
Specification animation	This involves program (or specification) execution as a way to validate the formal specification. It is similar to testing

However, the mathematics involved in most formal methods is reasonably elementary, and, in theory, if both customers and programmers are willing to learn the formal mathematical notation, then a rigorous validation of the formal specification can take place to verify its correctness. Both parties can review the formal specification to verify its correctness, and the code can be verified to be correct with respect to the formal specification. It is usually possible to get a developer to learn a formal method, as a programmer has some experience of mathematics and logic; however, in practice, it is more difficult to get a customer to learn a formal method.

This means that often a formal specification of the requirements and an informal definition of the requirements using a natural language are maintained. It is essential that both of these documents are consistent and that there is a rigorous validation of the formal specification. Otherwise, if the programmer proves the correctness of the code with respect to the formal specification, and the formal specification is incorrect, then the formal development of the software is incorrect. There are several techniques to validate a formal specification (Table 18.3) and these are described in [20]:

Why are Formal Methods difficult?

Formal methods are perceived as being difficult to use and of offering limited value in mainstream software engineering. Programmers receive some training in mathematics as part of their education. However, in practice, most programmers who learn formal methods at university never use formal methods again once they take an industrial position.

It may well be that the very nature of formal methods is such that it is suited only for specialists with a strong background in mathematics. Some of the reasons why formal methods are perceived as being difficult are (Table 18.4)

Characteristics of a Usable Formal Method

It is important to investigate ways by which formal methods can be made more usable to software engineers. This may involve designing more usable notations and better tools to support the process. Practical training and coaching to employees can help also. Some of the characteristics of a usable formal method are (Table 18.5).

Table 18.4 Factors in difficulty of formal methods

Factor	Description
Notation/intuition	The notation employed differs from that used in mathematics. Many programmers find the notation in formal methods to be unintuitive
Formal specification	It is easier to read a formal specification than to write one
Validation of formal specification	The validation of a formal specification using proof techniques or a Fagan like inspection is difficult
Refinement[a]	The refinement of a formal specification into successive more concrete specifications with proof of validity of each refinement step is difficult and time consuming
Proof	Proof can be difficult and time consuming
Tool support	Many of the existing tools are difficult to use

[a]It is highly unlikely that refinement is cost effective for mainstream software engineering. However, it may be useful in the regulated environment

Table 18.5 Characteristics of a usable formal method

Characteristic	Description
Intuitive	A formal method should be intuitive
Teachable	A formal method needs to be teachable to the average software engineer. The training should include (at least) writing practical formal specifications
Tool support	Good tools to support formal specification, validation, refinement and proof are required
Adaptable to change	Change is common in a software engineering environment. A usable formal method should be adaptable to change
Technology transfer path	The process for software development needs to be defined to include formal methods. The migration to formal methods needs to be managed
Cost[a]	The use of formal methods should be cost effective with a return on investment. There should be benefits in time, quality and productivity

[a]A commercial company will expect a return on investment from the use of a new technology. This may be reduced software development costs, improved quality, improved timeliness of projects or improvements in productivity

18.16 Review Questions

1. What are formal methods and describe their potential benefits? How essential is tool support?
2. What is stepwise refinement and is it realistic in mainstream software engineering?
3. Discuss Parnas's criticisms of formal methods and discuss whether his views are justified.

4. Discuss the applications of formal methods and which areas have benefited most from their use? What problems have arisen?
5. Describe a technology transfer path for the potential deployment of formal methods in an organization.
6. Explain the difference between the model-oriented approach and the axiomatic approach.
7. Discuss the nature of proof in formal methods and tools to support proof.
8. Discuss the Vienna Development Method and explain the difference between standard VDM and VDM♣.
9. Discuss Z and B. Describe the tools in the B-Toolkit.
10. Discuss process calculi such as CSP, CCS or π–calculus.

18.17 Summary

This chapter discussed formal methods, which are a rigorous approach to the development of high-quality software. Formal methods employ mathematical techniques for the specification and formal development of software, and are very useful in the safety critical field. They consist of formal specification languages or notations; a methodology for formal software development; and a set of tools to support the syntax checking of the specification, as well as the proof of properties of the specification.

Formal methods allow questions to be asked and answered about what the system does independently of the implementation. The use of formal methods generally leads to more robust software and to increased confidence in its correctness. There are challenges involved in the deployment of formal methods, as the use of these mathematical techniques may be a culture shock to many staff.

Formal methods may be model oriented or axiomatic oriented. The model-oriented approach includes formal methods such as VDM, Z and B. The axiomatic approach includes the process calculi such as CSP, CCS and the π calculus.

The usability of formal methods was considered as well as an examination of why formal methods are difficult and what the characteristics of a usable formal method would be.

References

1. The Z Notation. A Reference Manual. J.M. Spivey. Prentice Hall. International Series in Computer Science. 1992.
2. Rational for the development of the U.K. Defence Standards for Safety Critical software. M.J. D Brown. Compass Conference. 1990.

3. Applications of Formal Methods. Edited by Michael Hinchey and Jonathan Bowen. Prentice Hall International Series in Computer Science. 1995.
4. 00-55 (Part 1)/ Issue 1. The Procurement of Safety Critical Software in Defence Equipment. Part 1: Requirements. Ministry of Defence. Interim Defence Standard. UK. 1991.
5. 00-55 (Part 2)/ Issue 1. The Procurement of Safety Critical Software in Defence Equipment. Part 2: Guidance. Ministry of Defence. Interim Defence Standard. UK. 1991.
6. The Evolution of Def Stan 00-55 and 00-56. An intensification of the formal methods debate in the UK. Margaret Tierney. Research Centre for Social Sciences. University of Edinburgh. 1991.
7. Experience with Formal Methods in Critical Systems. Susan Gerhart, Dan Craighen and Ted Ralston. IEEE Software. January 1994.
8. The Structure of Scientific Revolutions. Thomas Kuhn. University of Chicago Press. 1970.
9. Mathematical Approaches to Software Quality. Gerard O' Regan.Springer. 2006.
10. The Vienna Development Method. The Meta language. Dines Bjørner and Cliff Jones. *Lecture Notes in Computer Science* (61). Springer Verlag. 1978.
11. Formal Specification and Software Development. Dines Bjørner and Cliff Jones. Prentice Hall International Series in Computer Science. 1982.
12. Computation Models and Computing. PhD Thesis. Mícheál Mac An Airchinnigh. Dept. of Computer Science. Trinity College Dublin.
13. How to Solve It. A New Aspect of Mathematical Method. Georges Polya. Princeton University Press. 1957.
14. Proof and Refutations. The Logic of Mathematical Discovery. Imre Lakatos. Cambridge University Press. 1976.
15. MSc. Thesis. Eoin McDonnell. Dept. of Computer Science. Trinity College Dublin. 1994.
16. The Science of Programming. David Gries. Springer Verlag. Berlin. 1981.
17. A Disciple of Programming. E.W. Dijkstra. Prentice Hall. 1976.
18. Communicating Sequential Processes. C.A.R. Hoare. Prentice Hall International Series in Computer Science. 1985.
19. A Calculus of Mobile Processes. Part 1. Robin Milner et al. LFCS Report Series. ECS-LFCS-89-85. Department of Computer Science. University of Edinburgh.
20. A Personal View of Formal Methods. B.A. Wichmann. National Physical Laboratory. March 2000.
21. On the Criteria to be used in Decomposing Systems into Modules. David Parnas. Communications of the ACM, 15(12). 1972.