Part I

Why Z?
I Formal methods

Formal methods apply logic and simple mathematics to programming. They work best where traditional programming methods don’t work very well: problems that are too difficult to solve by intuition or too novel to solve by modifying some existing program or design. They can help you create new programs, or analyze and document programs that are already written. Using formal methods requires creativity and judgment, but once you have created or analyzed a program formally, you can document your work as a sequence of steps that you or anyone else can check. You must be able to do this if you need to convince yourself or others that a program meets requirements for safety, accuracy, security, or any other critical property. It is also worth doing if you simply want to understand how the program works.

1.1 What are formal methods?

Formal methods are methods that use formulas.

A formula is a text or diagram constructed from predefined symbols combined according to explicit rules. A good working definition of formula is anything whose appearance or syntax can be checked by a computer. According to this definition, every computer program is a formula.

It’s a little odd for programmers to speak of formal methods as if they were something special – as if formality were an option. If you want to program a computer, you really don’t have any choice. Computation is formula evaluation.

And yet, formal methods have become something to make a fuss over, something that many programmers are said to be unwilling – or unable! – to use. What distinguishes these formal methods from what programmers already do every day?

The special meaning of formal methods often appears on when we use formulas. When we’re doing formal methods, we don’t just write the code in a formal notation, we also use formal notations in the stages that come before coding. We express the specification or design in formulas.
Chapter 1. Formal methods

Formal methods are also distinguished by what we use the formulas to express: the behavior of programs. Many programmers only know one way to determine the meaning or semantics of a formula: execute it on a computer. With formal methods we can determine the meaning of a formula – be it a specification, design, or program – without executing it. The whole point of using formal methods is to be able to predict what a program will do without running any code – in fact, without writing any. This means we can discover many errors without having to run any tests. This ability to model behavior distinguishes Z and other formal notations from diagramming methods that can only model program structure.

Formal methods are also distinguished by the choice of notation. The formal notations we use to express specifications and designs are usually different from our programming language – we can use a distinct specification language. The specification language need not be executable and may resemble traditional mathematics and symbolic logic more than a programming language. These languages can be more expressive, more concise, and easier to understand than any executable programming language, and they usually come with a lot less syntactic clutter. Z is one of these nonexecutable specification languages (some alternatives are surveyed in Appendix G). We usually call Z a notation rather than a language to emphasize its mathematical nature.

Formal notations such as Z are distinguished from less formal notations such as data flow diagrams because they have a formal semantics that assigns a precise meaning to any formula in the notation. Moreover, a formal notation comes with laws that enable us to simplify formulas, derive new ones, and determine whether one formula is a consequence of others. This is what most distinguishes mathematical notations such as Z from natural human languages and also from most programming languages. It makes it possible to derive designs and code from a specification, and to check whether code and designs correctly implement a specification. Moreover, since formal notations can be processed by machine, parts of these tasks can be automated.

Formal methods are a kind of analysis. Analysis is any activity devoted to understanding software without actually running programs, including reviews, inspections, and walkthroughs – anything that involves reading, discussing, and trying to understand programs without testing. Analysis can be more effective than testing for many purposes, because you can analyze an entire program text, but you can only test a (usually very small) sample of program behaviors. Many studies have found that informal analyses can be more effective than testing for detecting errors and improving software quality [Fagan, 1986; Ackerman, Buchwald, and Lewski, 1989; Russell, 1991; Knight and Myers, 1993]. What we call formal methods are just particular kinds of analyses that employ mathematical notations.

Formal methods can help you create software so that you can understand it before you run it. You shouldn’t have to resort to guessing to produce programs. You needn’t rely on trial and error to validate and improve them. You still need to test, but it no
1.2. What formal methods are not

longer serves as your primary error detection method. You make sure that the code
is right by being sure about all the stages leading up to it.

1.2 What formal methods are not

Formal methods are not project management methods. People in computing often use
the word *formal* rather loosely, to mean strict, detailed, or methodical. Sometimes
formal even connotes a particular style of doing software projects that involves
following a lot of written procedures that are enforced by management. In this
book, I reserve formal for methods that use logical and mathematical formulas\(^1\).
Management methods are concerned with the process used to create the program,
but with formal methods we can assess the program directly.

Another common misconception is that formal methods use one particular mod-
elling technique that is in competition with other popular techniques. This mistake
is revealed by questions such as, “Do you use formal methods or object-oriented
programming?” Those are not mutually exclusive categories. You can use formal
methods with any modelling technique.

1.3 When are formal methods useful?

Formal methods involve writing another formal description of the program, in ad-
dition to the code itself. This might seem like extra work and it isn’t always useful.
Formal methods can help with novel projects, difficult projects, and critical projects.

Novel projects involve building something substantially new, where we can’t just
take an existing system and make some obvious modifications. We need to compare
design alternatives, not just plunge ahead and implement the first idea that comes
along. We can’t afford to build several versions of the whole system, so we have to
analyze models instead.

Projects are difficult when they tackle problems that are profound and deep, or
when they present a multitude of intricate details. Difficult projects need not be large;
a single page of code can present so many choices that trial-and-error guessing and
testing might never converge to a useful solution. We don’t have to throw up our
hands and complain how incredibly complicated it is. We can use formal methods
to derive a solution and check that it is correct.

\(^1\) However, the first definition of “formal specification” in an IEEE standard is “a specification written
and approved in accordance with established standards” [IEEE, 1987]. A recent book on avionics
states, “Formal methods are institutionalized procedures that permit managers, engineers and cus-
tomers to verify that development is proceeding without major problems” [Neuport, 1994].
Chapter 1. Formal methods

Most software is produced with the expectation that users will discover errors that the developers missed, but this is unacceptable for critical projects. Critical projects are those where so much is at stake – safety, security, money – that stakeholders outside the development group demand to be informed about the technical content of the product. These stakeholders might be management, customers, an external quality assurance organization, or government regulators. They are not content to discover how the product behaves in the course of using it; they require a detailed statement of what it will do – and what it will not do. They require the developers to show that the promised behavior has been achieved. A formal specification can describe the product behavior, and a formal development can help make the case that the product meets its specification.

Many programming projects are neither novel, difficult, nor critical. In truth, many of these are too tedious to be easy – let’s call them routine. Experienced programmers can adapt an adequate solution from their files, or their heads. Routine errors arise from fatigue, haste, or simple carelessness, and can be detected by inspection of the code against prose requirements – or implicit understandings that are not even written down. Once the program is running, it is easy to determine if the results are correct, and if they are not they can simply be discarded. In such routine jobs there is no need to use formal methods.

1.4 How can we use formal methods?

We use formal methods in three essential activities: modelling, design, and verification.

1.4.1 Modelling

Models enable us to describe and predict program behavior.

Many programmers believe that the only really accurate description of what a program does is the program text: the code itself. However a mathematical model can describe program behavior accurately and comprehensively, and it is often much shorter and clearer than the code. We can use the model to calculate or infer the behavior of the program before we code it. Modelling makes the behavior of the program predictable – a good property for any program to have, an essential property for a safety-critical system.

Complex systems can have surprisingly simple models. Finding the right model can be the key to a clear design and a compact, efficient program. The chapters in Parts II and IV present a series of models expressed in Z.

A model is a simplified representation. Computing confronts us with a mass of detail; models help us cope. A model leaves something out – it has some of the
1.4. How can we use formal methods?

properties of the system it models, but not all of them. We construct a model to focus on some particular aspect of a system, and we omit all the details that are inessential to that aspect. There can be several models of the same system, each focusing on a different aspect.

For example, a prototype that demonstrates the look and feel of a new system to prospective users is a kind of model because it does not provide all of the functions of the final product. It is a bit like an architect’s scale model or an artist’s rendering of a new building, because it is intended to convey an aesthetic impression. Sometimes we need a different kind of model, intended to represent the system’s functional behavior or internal structure. This is more like the mathematical models that structural engineers use to check that the beams will fit together and bear the loads. Z is used to create this latter type of model.

A mathematical model that represents the intended behavior of a program can be used as a formal specification. Programmers sometimes act as if formal specifications were a strange new idea. In fact, we have always used mathematical models in computing.

Here is an example from my own work. In radiation therapy we use computer programs to estimate the radiation dose distribution that would be created in the patient’s body by a proposed treatment [Khan, 1984]. Figure 1.1 shows part of the formal specification for our program [Kalet et al., 1993]; the notation is ordinary mathematics. It is supplemented by a picture to help illustrate the definitions of the variables that appear in the formula (not shown here are several pages of prose and formulas that also explain those definitions).

This next example may be more familiar to programmers. Figure 1.2 shows the formal specification for the syntax of numbers in the programming language Pascal. For example, Figure 1.2 permits 1, 0.5, 1E5, and 1.5E-5, but prohibits 1., .5, E5, 1.5E-5.0, and so forth. Figure 1.2 appears in the language reference manual [Jensen and Wirth, 1974] to help programmers understand how to form numbers. However, it can also be considered part of the formal specification for a Pascal compiler. The formal notation is called Backus-Naur Form (BNF). Here again, there is a picture. It illustrates an alternative view of the information presented in BNF. The entire syntax of the Pascal language (which describes every syntactically correct Pascal program) is given in five pages of BNF. Thanks to formal models like this one, writing a correct syntax analyzer for a compiler is a straightforward task. This achievement belies the programmers’ complaint that hard problems present too many cases to anticipate. Any compiler can handle a virtually infinite number of distinct cases (program texts), accepting all syntactically legal programs and rejecting every illegal one.

These examples should remind you that using formal specifications is really not such a strange thing to do. We already use formal specifications in complex applications where we know how to write substantially correct programs that people
8 Chapter 1. Formal methods

The dose $D$ at a point $P$ inside the patient (see figure 1) due to a single fixed external photon or neutron beam source with a rectangular collimator is calculated by the formula

$$D = \left(\frac{F}{F + m}\right)^2 \cdot D_{\text{cut}} \cdot O(w_2) \cdot \left[\frac{T_{PR}(w_1d, d) \cdot OCR\left(w_1, d, \frac{2x}{w_1}\right) \cdot OCR\left(w_2, d, \frac{2y}{w_2}\right)}{B}\right] \cdot I \cdot W$$

(1.1)

where $D$ is dose per machine unit at point $P$ within the patient.

Figure 1.1: Formal specification for a radiation dose calculation program

can understand. It would be crazy to try to write either of these programs without the formal specification. Can you imagine trying to write a compiler where the syntax was only defined by a lot of prose and examples describing particular special cases?

Can you imagine trying to use a compiler that was written that way? No competent programmer today would even consider such a thing, and no physicist would begin coding a calculation without the formula close at hand.

These examples also show that a mathematical model does not much resemble a prose description. It is no mere paraphrase of the prose into another notation; it is a different expression of the same behaviors, in a form that is better organized to serve as a guide for programming. Bridging from the users' informal view of the
1.4. How can we use formal methods?

Figure 1.2: Formal specification for Pascal number syntax.

requirements to the programmers’ formal model is one of the central creative tasks in programming. This task is discussed in Chapter 3.

In both examples, the formal model enables us to calculate the results the program should get. We can use the formal model as an oracle, an independent standard of accuracy that can tell us what the result of executing the program on any test case should be. We can use the model to help us choose test cases and tell whether the program passed the tests.

An oracle can also help us determine whether we made the right decisions about what the program should do — that is, whether we got the requirements right. Failure to understand the real requirements has been responsible for many software failures and accidents, and these errors can be the most expensive and difficult to fix. Because a formal model enables us to predict program behavior, we can investigate how our program would behave even before we begin design and coding. If we have a formal model, we can apply powerful analytical techniques to confirm that it meets critical requirements such as safety or security. Chapter 15 describes some techniques used in those analyses.

Programmers often try to write complex programs with no formal model; the only formal description is the code itself. As a result the expected system behaviors and the assumptions about the environment cannot be reviewed, criticized, or even
Chapter 1. Formal methods

examined because they are hidden in the program text itself, usually encoded in some obscure way — they don’t call it code for nothing. Such a system is not predictable; the only way to really find out what it does is to experiment with it. This is unacceptable for systems that have requirements for safety, security, accuracy, or any other critical property.

A formal specification can be valuable even when the rest of the development is informal, because it provides an oracle for analyzing requirements and planning tests.

1.4.2 Design

Design means organizing the internal structure of a program.

There are two dimensions to design: partition and refinement. Partition means dividing the whole system into parts or modules that can be developed independently. Refinement means adding detail, going from an abstract model that clearly satisfies the original requirements to a concrete design that is closer to code\(^2\).

Many informal software development methods address design. Most of them teach a particular way to draw and annotate diagrams that you can use to document designs, such as bubble-and-arrow data flow diagrams. Some of these methods are supported by software products called CASE tools to help you produce the diagrams and documents. However, the formal content of many of these methods is weak. They can only represent the structure of a program: what the program’s parts are and how they are related to each other. They provide few criteria to determine whether a design is correct, or to choose the best of several plausible designs.

\(Z\) is a more powerful design notation because it can also model behavior. Finding the best structure usually depends on understanding the behavior. In \(Z\) you can express which components of the system are needed to perform any behavior, down to any level of detail you need (even to individual program variables). This enables you to see how components must be grouped together to provide the behaviors you need. You can find the best way to partition your system into modules.

\(Z\) can also support constructive approaches to design. Rather than work top-down and partition an abstract specification into modules that we have to implement ourselves, we might achieve savings by working bottom-up and assembling our system from prefabricated building blocks or reusable software components. In this enterprise — “programming in the large” [DeRemer and Kron, 1976] — the problem shifts to identifying which blocks to use and determining how they should fit together. \(Z\) can express precise descriptions of what the blocks do and enable us to calculate how the whole system will behave when the blocks are used in combination. Chapter 15 describes how to infer properties of systems described in \(Z\).

\(^2\) In some formal methods literature, refinement is called reification.
1.5. Are formal methods too difficult?

Z can also help you determine whether your design will work. Mathematics can tie development stages together in a way that is not possible with informal notations. It provides the only way we have to show that an abstract model and a detailed design are two views of the same thing. If we have a mathematical model, we can infer development steps, and then we can check each step by calculation and proof. This means we need not rely solely on intuition to tell whether our designs are correct. We can find and correct design errors without coding and testing a program. Chapters 15 and 26 describe how to check designs.

1.4.3 Verification

Verification means showing that our code will do what we intend.

Verification deals with the final product of our development: code in some executable programming language. One of the products of a formal verification is a proof, a convincing demonstration — based only on the specification and the program text, not on executing the program — that the code does what its specification requires. Proof can provide greater confidence than testing because it considers all cases, while testing just samples some of them. Moreover, proof can be more convincing than appeals to intuition because it can be more explicit, easier to check, and therefore less fallible than intuition.

Much of the early research in formal methods concentrated on formal verification. Some even experimented with automating the proofs. This early work was so influential in fixing the image of the field, that when you say formal methods, many programmers still think, “proving the correctness of code,” or even “automated proof.” This perception is no longer accurate; much recent work concerns modelling and design, in addition to verifying code.

Chapter 27 discusses formal verification and shows that the most efficient way is usually to do the verification in the course of deriving a program from a formal specification.

1.5 Are formal methods too difficult?

Using formal methods can be more difficult than programming in the usual way — because formal methods aim higher. Describing exactly what your program does is more difficult than letting testers or users figure it out for themselves. Making your program do the right thing in every situation is more difficult than just handling some typical cases. Any method that can handle hard problems will sometimes be hard to carry out; only superficial methods can be easy all the time.

Fortunately, most of the mathematics we need for formal methods is not terribly difficult. The discrete mathematics used in this book — and in most practical appli-