Chapter 0

Introduction

. . . for by the error of some calculator the vessel often splits upon a rock that should have reached a friendly pier . . .

- Henry David Thoreau.

We all know from personal experience that computers do not work correctly all the time. For most of us this realisation manifests itself with nothing more serious than delays and frustrations as we encounter automatic bank tellers which are out-of-order or Web sites which are faulty, or face long waits at airports as glitches in the booking, check-in, or even the flight control system are being catered for.

However, the problems of systems failures become more serious (costly, deadly) as automatic control systems find their way into almost every aspect of our daily lives. It is recognised – and accepted – that complete reliability of any major software system is beyond expectation. While, for example, civil and mechanical engineers can build impressive bridges which are guaranteed to remain standing, and aeronautical engineers can design aeroplane wings which behave in precise and predicatable ways, Software Engineers are almost never so successful. Computers carry viruses, hang, crash or die; and their software is full of security leaks and bugs. Designing dependable high quality computing systems remains a challenge for Software Engineers.

Quality expectations, of course, have much to do with culture and context. Many of us are willing to accept as a mere inconvenience that a train can be delayed, a cash machine can be out of order, or a mail server can temporarily be down. But we don't tolerate bridges that fall down, nuclear meltdowns in power stations, or security leaks in public data bases. Engineers speak about *safety critical* applications when system failure cannot be tolerated, but we should expect software systems to be user friendly, safe and dependable in any application context. Why is this so difficult to achieve?

There are several answers to this question. One of them is that software systems can be extremely complex – more complex even than most other systems that Engineers can design and build. They may consist of large numbers of heterogeneous components that can change over time and interact

F. Moller, G. Struth, *Modelling Computing Systems*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-1-84800-322-4_1, © Springer-Verlag London 2013

in intricate and sophisticated ways. Another answer is that our knowledge and experience in designing such systems, and our expertise in organising their design process, are still in their infancy compared to building bridges, chemical plants or railway networks. A third answer – and perhaps the most important one – is that rigorous mathematical tools and methods are much less employed in Software Engineering than in other engineering disciplines. While traditional engineers have always been academically trained to use tools and techniques from mathematics and physics to guarantee the quality of their products, software designers and programmers are still often self-taught and have traditionally relied very much on their intuition and intelligence. Many of them seem to have a rather fatalist attitude towards bugs.

This attitude is more and more difficult to defend. Programs and software systems are themselves mathematical structures; many software systems rely on sophisticated mathematical mechanisms such as audio compression, public key cryptography, or Web search ranking; and there are powerful domain-specific mathematical tools and techniques that can help us to understand, design, implement and analyse them in a better, more structured, and more scientific way.

The aim of this book is to introduce some of the techniques which, when applied, can help to reduce the number of errors present in a system. Errors can arise at many points in the software development process, from understanding exactly the requirements and behaviour of the system being built, to ensuring that these requirements are correctly captured in the design and implementation of the system. By working within the confines of a precise structured method, the occurrence of such errors can be drastically curtailed.

0.1 Examples of System Failures

To understand and appreciate the role of mathematics in modelling computing systems, it is helpful to look at a variety of examples of system failure. Some of these failures are of an entirely technological nature, others have to do with the ways in which humans and machines interact or in which rules of communication between different agents have been designed. In every case, they arise from errors in information processing, which is at the very core of computational modelling.

0.1.1 Clayton Tunnel Accident

Up until the mid-nineteenth century, collisions between trains were avoided solely by enforcing a minimum time interval between trains. Railway em-

ployees (known as "policemen") would stand at regular intervals ("blocks") along the line and signal trains with hand gestures to slow down if too little time had elapsed since the previous train had passed. In the case of a break-down of a train, the guard in the rear of the train would run back along the track to warn any oncoming trains of the danger ahead.

With ever-increasing traffic, growing lapses in this system eventually led to the installation of crude *block signalling* in particularly troublesome places, in which some *protocol* would be followed to ensure that only one train occupied a given stretch of track at any given moment. Such a protocol typically involved railway workers at each end of the section signalling each other via telegraph of the passage of trains: having let one train proceed, the signalman would hold back any further trains until a message was received indicating that the first train had cleared the section ahead. The first commercial electric telegraph was constructed in Britain for use on the Great Western Railway, and the first section of track to be protected by block signalling using telegraph communication was the track through Clayton tunnel outside Brighton. However, on the morning of Sunday 25 August 1861, this protocol failed to prevent a catastrophic collision inside the tunnel which killed 23 people and injured a further 176.

In normal operation, when a train arrived at Clayton tunnel, it would meet a rail-side signal which would be set at "danger" unless the signalman at the entrance to the tunnel set it to "all right" authorising the train to enter the tunnel. This signalman would telegraph a "train in tunnel" message to the signalman at the other end of the tunnel, and the rail-side signal would be reset to "danger" to prevent any further trains from entering the tunnel until the signalman received a "tunnel clear" message by telegraph from the signalman at the other end of the tunnel, indicating that the train had emerged from the other side.

On the fateful morning in question, three trains left Brighton Station within a seven-minute period and steamed towards Clayton tunnel. These trains were scheduled to depart at 8:05, 8:15 and 8:30, respectively; however, the first train was running late, and the assistant stationmaster in charge that morning – one Charles Legg – opted to ignore the strict regulation of ensuring a minimum five-minute separation between trains by sending them off at 8:28, 8:31 and 8:35, respectively. The first train was given the "all right" signal to enter the tunnel, and the signalman – named Henry Killick – telegraphed the "train in tunnel" message to his counterpart – a man by the name of Brown – at the other end. He was then taken by surprise by the quick arrival of the second train, which passed the rail-side signal before he had had a chance to reset it to "danger." In desperation, he rushed out of his cabin furiously waving his red flag to stop the second train just as it was disappearing into the tunnel; there was no way for him to know, however, whether or not the driver had seen the flag.

Killick telegraphed to Brown a further "train in tunnel" message and waited tentatively for a response. Killick telegraphed a further message to Brown asking if the tunnel was clear; and to his relief he finally received the "tunnel clear" message from Brown. Unfortunately for Killick, Brown had not realised from Killick's repeated "train in tunnel" message that a second train had entered the tunnel; his "tunnel clear" message was in response to the passing of the first train. The driver of the second train – unbeknownst to Killick – had in fact seen the red flag; and having finally brought his heavy load to a stop, he was in the process of cautiously reversing back towards Killick. When, at that moment, the third train arrived at the entrance to the tunnel, Killick offered it the "all right" signal – with fatal consequences.

The Clayton tunnel accident is obviously not the result of a computer failure, but it is based on a poorly designed communication protocol between distributed agents, and therefore typical for computing systems. *Mutual exclusion algorithms,* which prevent more than one computing agent at a time accessing a resource such as a printer or a global variable, are instrumental parts of any operating system. The accident also shows a standard pitfall of systems design: the whole idea of the signalling protocol at Clayton tunnel was to ensure that two trains could not occupy the same block at the same time. But it couldn't handle the exceptional case it was supposed to prevent. (For details, see L.T.C. Rolt, *Red for Danger: The Classic History of Railway Disasters*, The History Press, 2009.)

0.1.2 USS Scorpion

In 1968, the nuclear submarine USS Scorpion was destroyed killing all of its 99 crew members. Though the cause of its destruction has long been steeped in mystery, evidence which was only declassified three decades later suggests that the submarine may in fact have been destroyed by one of its own torpedoes which had been accidentally activated and thus ejected. The torpedo had been cleverly designed to seek out its nearest target, which is precisely what it did on this occasion, with devastating consequences. (For details, see P G Neumann, *Computer Related Risks*, Addison Wesley, 1994.)

The negative implications of seemingly sensible and harmless design decisions often arise only in hindsight as unintended consequences after disaster has struck. Clearly, every eventuality needs to be accounted for, especially in safety-critical designs where failure of the system could lead to injury, illness or loss of life; serious environmental damage; or major financial loss.

0.1.3 Therac 25 Radiotherapy Machine

The Therac 25 was a radiation therapy machine that intermittently gave the wrong radiation doses over a period of three years (1985-87) due pre-

dominantly to errors in the software controlling its operation, as well as its poor interface design. The problems with the Therac 25 have been very thoroughly analysed, and six accidents – three of them fatal – have been attributed to its failures. (For details, see N Leveson and C S Turner, "An Investigation of the Therac-25 Accidents," *IEEE Computer* **26**(7), pages 18–41, July 1993.)

The basic issue involved the replacement of hardware interlocks used in previous models by a software-only system. The machine had two modes of operation: electron mode and photon (or X-ray) mode, which were used for treating tumours at different depths in a patient's body. Electron mode involved a low-power electron beam, while photon (X-ray) mode involved a high power electron beam (three orders of magnitude more powerful), but with a metal plate between the device and the patient, to generate the X-rays. The electron beam had to be in low-power mode if the plate was not present, and in earlier designs (Therac 6 and Therac 20) there was a mechanical interlocking device which physically ensured this. This hardware interlock was removed from the Therac 25 which was left to rely on a (faulty) software interlock.

The software was poorly specified (there was no documentation on its software specification), designed and tested; and much of it was imported as-is from the previous models despite changes in requirements, without any form of integration testing. The problem was compounded by a complex user interface. In some cases, if the operator tried to enter certain control sequences (either in error or as shortcuts), the machine would operate incorrectly, using the high-power beam with no plate. It would then report an error, which it would normally do when no treatment had been delivered. Often in response to such an error report, operators would repeated the whole process, leading ultimately to fatal unintended consequences.

Leveson and Turner draw the following conclusion: "Virtually all complex software will behave in an unexpected or undesired fashion under some conditions – there will always be another bug. Accidents are seldom simple – they usually involve a complex web of interacting events with multiple contributing technical, human, and organisational factors." To improve the situation, they appeal to education: "Taking a couple of programming courses or programming a home computer does not qualify anyone to produce safety-critical software." The lesson is clear: the same rigorous standards should be applied to Software Engineering as to Engineering in general.

0.1.4 London Ambulance Service

In October 1992, the London Ambulance Service installed a computer aided dispatch (CAD) system, known as LASCAD, to control the dispatching of ambulances across London. It was to automatically match up each call to be responded to with the closest available ambulance. However, the system

was unable to cope with real-time data, which was on the order of 5000 calls per day. As it became more and more swamped with information and requests, it generated more and more exception messages requiring human intervention. The volume of these messages caused the exception messages, together with information needing to be dispatched to ambulances, to scroll off the top of the controllers' screens. As many as thirty deaths have been attributed to failings of the system. For example, it was reported that one ambulance arrived to find the patient had died and long since been collected by the undertaker; and that another ambulance took 11 hours to reach its destination – five hours after the stroke victim had made their own way to the hospital.

The London Ambulance Service quickly reverted partially to its manual dispatching system. However, after eight days, the automated system crashed completely, leading the service to revert completely to the original manual system. Taking responsibility for the £1.5 million failure, the chief executive of the London Ambulance Service duly resigned from his post. (For details, see A Finkelstein and J Dowell, "A Comedy of Errors: The London Ambulance Service Case Study," in *The Eighth International Workshop on Software Specification and Design*, IEEE CS Press, pages 2-4, 1996.)

As in our previous examples, this disaster was caused by a complex web of managerial and economic pressure, incompetence and technical failure; but Finkelstein and Dowell conclude that "at the heart of the failure are breakdowns in specification and design common to many software development projects."

0.1.5 Intel Pentium

When the Intel Pentium PC was initially released in 1994, problems were found in its floating-point unit. With certain inputs, the unit gave inaccurate results when performing division, thus rendering it useless for mathematical or scientific work.

The error had been caused in the design stage of the chip when a new algorithm for floating-point division was implemented which was three to five times faster than previous methods. This algorithm is based on using look-up tables to calculate intermediate results. The hardware was implemented using a program to download values into the look-up tables; however, an error in this software caused five of the 1066 entries to be inadvertently omitted.

Because the calculations recursively use information from the look-up tables, the errors that can accrue magnify in scale. For example, performing the sum $x - (x/y) * y$ should return the answer 0 for any inputs x and y. Given that computers have to deal with approximations to real numbers, we typically have to settle for a value close to zero to be returned. But

with input values $x = 4195835$ and $y = 3145727$ the first Pentium release gave the answer 256. (For details, see T R Halfhill, "The Truth Behind the Pentium Bug," *Byte* 20(3), pages 163-164, March 1995.)

0.1.6 Ariane 5

In June 1996, the maiden flight of the Ariane 5 satellite launch rocket, Flight 501, ended in disaster: the rocket veered off course and exploded 40 seconds after lift-off. Its self-destruct system was initiated when the rocket detected it was disintegrating. This damage was caused by friction with the atmosphere as the rocket was travelling at too shallow an angle.

The flight path of the rocket was controlled by two software components, one providing the flight data, and the other converting this data into signals which controlled nozzles that direct the rocket's boosters. The problem was found to be with the software providing the flight data, which was imported as-is from the earlier Ariane 4 (a similar problem underlying the Therac 25 failure).

The software executed an instruction to convert a 64-bit integer to a 16-bit representation on a number that was too big to be stored as a 16-bit integer. (Ariane 5 used a different flight path from Ariane 4 which involved a shorter period of vertical ascent before yawing over to accelerate, thus reaching shallower angles than Ariane 4 sooner in the flight; this problem thus never arose with Ariane 4.) As there was no code to deal with this exception, the program crashed, and the ensuing error messages generated by the system were interpreted by the guidance system as flight data. Ironically, the part of the software that failed was only needed by Ariane 4 before lift-off, and was only active during the first part of the flight due to the possibility of a short hold prior to lift-off. This piece of software was unnecessary for Ariane 5.

The Ariane 501 Inquiry Board reported that the failure was "due to specification and design errors in the software of the inertial reference system" because the Ariane 5 Development Programme "did not include adequate analysis and testing of the inertial reference system or of the complete flight control system." It recommended that the European Space Agency should in the future ascertain that "specification, verification and testing are of consistently high quality." (For details see "Ariane 501 Inquiry Board report," `http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf`.)

0.1.7 Needham-Schroeder Protocol

When communicating over the Internets, where anyone can intercept and read the messages you send, it is important to securely encrypt any sensitive information that you may send out, such as your credit card details, so that only the intended recipient of your message can decrypt and read it. The

Needham-Schroeder protocol was devised to allow two parties – commonly referred to as "Alice" and "Bob" – to authenticate themselves over such an insecure channel: after executing such a protocol, Alice will believe that she is talking to Bob and vice versa, and hence they will have established mutual trust for further transactions.

The Needham-Schroeder protocol is based on *public-key cryptography*: Bob (and anyone else) can, for instance, use Alice's public key – which he can obtain from some trusted server – for encrypting messages to Alice which only Alice can decrypt and read using her private key which she keeps secret. The protocol then works as follows:

1. Alice sends a message to Bob – encrypted with his public key – consisting of a random number along with some statement about her identity.
2. Bob decrypts this message with his private key, and sends a message in response to Alice - encrypted with her public key – consisting of Alice's random number along with a random number of his own.
3. Alice decrypts Bob's message with her private key, and sends another message to Bob – again encrypted with his public key – consisting of Bob's random number.

When Alice receives Bob's response to her first message, she will believe that she is talking to Bob, as only Bob could have decrypted her message and discovered the random number that she had sent him. Equally, when Bob receives Alice's second message, he will believe that he is talking to Alice, as only she could have decrypted his message and discovered the random number that he had sent her. Hence, after executing this protocol, Alice and Bob will both have reason to trust each other's identities.

This protocol was devised in 1978, and for over 15 years it gave no cause for concern to the network community. Indeed, there were a variety of "proofs" attesting to the correctness and reliability of this protocol. Despite this evidence of the protocol's security, in 1995 it was discovered to be susceptible to a very basic *man-in-the-middle attack*: an intruder could participate in the protocol and convincingly impersonate another agent – even without breaking the encryption. Here is how it works:

1. The intruder masquerades as Bob so that Alice encrypts her initial message with the intruder's public key and sends her message to him.
2. The intruder decrypts Alice's message with his private key, then encrypts it with Bob's public key and sends this on to Bob.
3. Bob sends Alice's random number together with his own, encrypted with Alice's public key, to the intruder, who forwards it – unaltered – to Alice.
4. Alice decrypts Bob's message, encrypts Bob's random number with the intruder's public key and sends it to the intruder.

5. The intruder decrypts this message, encrypts it with Bob's public key and sends this on to Bob.

As far as Alice and Bob are concerned, the results of this interaction as interfered with by the intruder appear identical to those of the original interaction, so they will once again believe – this time incorrectly – that they are talking directly to each other. Their subsequent correspondence will all be via the intruder, who will be able to read all of Alice's messages, as they will continue to be encrypted using the intruder's public key and re-encrypted by the intruder with Bob's public key before being sent on to Bob. The intruder will still not be able to read Bob's messages, though, as these will all be encoded using Alice's public key, and the intruder will only be able to forward these unaltered to Alice. (For details, see G Lowe, "An attack on the Needham-Schroeder public key authentication protocol," *Information Processing Letters* 56(3), pages 131-136, November 1995.)

In contrast to the previous examples, this is a pure design error, which is again rather unexpected and surprising given the simplicity and stringency of the protocol. The difficulty with detecting this flaw is that intruders can behave in various unexpected ways that – being unpredictable – are very difficult to analyse. Even very simple protocols can lead to a wide variety of different system behaviours that need to be considered. It seems rather improbable that such diversity can be catered for simply by testing.

The various failures discussed above have complex and generally multiple causes, and most of them can be traced back to poor software development processes. What is lacking in the development process is a rigorous engineering discipline through which a thorough understanding of the system being developed is obtained before the system is constructed. In traditional engineering disciplines, the methods for obtaining such an understanding are well established and based on formally modelling an appropriately-abstract version of the system being developed. The challenge for Software Engineering is to mimic these methods; to do so requires an understanding of how to describe and analyse abstract models of software systems. Of course, this first requires an understanding of these terms.

0.2 System, Model, Abstraction and Notation

The notions of "system," "model," "abstraction" and "notation" are essential to this book. In this section we provide various dictionary-style definitions of these concepts, interspersed with some examples and thoughts.

System

> An assemblage of objects arranged in a regular subordination, or after some distinct method, usually logical or scientific; a complete whole of objects related by some common law, principle, or end; a complete exhibition of essential principles or facts, arranged in a rational dependence or connection; a regular union of principles or parts forming one entire thing.

How do we understand systems and put them together? In the object oriented approach to software design, one is guided by the above dictionary definition and methodically describes the whole by giving descriptions of the constituent parts along with how these parts are put together. If you have tried and trusted building blocks, then you can reliably use them again.

To understand and analyse the world in terms of systems is very important to science; and to build them is the fundamental task of Engineers. Systems can be described, for instance, in terms of their *structure* – how they can be decomposed into parts and how these parts are related to each other; or in terms of their *behaviour* – how they evolve and interact with their environment; or in terms of their *functionality* – what their goals and objectives are. Systems are often contrasted with the environments in which they are embedded and with which they interact. A prime example from the world of computers is the *operating system*, which manages our interactions with the computer hardware.

This book addresses *computing systems*. However, we understand the term "computing" in a rather loose sense. We do not identify computing systems with computers, but with all kinds of systems that access, store, process and communicate information. Many biological, physical, economical and social systems have recently been studied from this point of view, and many of the concepts and techniques introduced in this book can be used in these contexts.

Model

> (1) A miniature representation of a thing, with the several parts in due proportion; sometimes, a facsimile of the same size. (2) Something intended to serve, or that may serve, as a pattern of something to be made; a material representation of or embodiment of an ideal; sometimes a drawing or a plan; a description of observed behaviour, simplified by ignoring certain details.

Building models is at the core of any scientific and engineering discipline. Scientists need models to interpret their data and make predictions; and traditional engineering products such as bridges and aeroplanes are never built until models of them have been developed and studied to understand

the characteristics of the product. These models may be small-scale versions of the product which are tested in wind tunnels; or they may be purely abstract models described on paper using some formal notation which are then analysed more formally, for instance through simulations on a computer.

Modelling techniques are also becoming more and more important in software engineering, as computing systems become ever more complex and ubiquitous, and their proper functioning is often extremely critical. It is no longer possible to rely on the cleverness of our programming skills when building computing systems. In this book we shall explore basic modelling techniques for software engineering; consider various simple illustrative yet sufficiently interesting computing systems; and describe models that capture those aspects of their behaviour that interest us.

Models come in all shapes and sizes, and are designed to capture specific aspects of the thing they represent. Consider, for example, the following two uses of simple railway models.

- If we are interested in teaching the history of the development of railway locomotives, then full scale working replicas would be fun, but probably inconvenient; small scale working replicas might do, or even non-working replicas. Meaning (1) is appropriate.
- If we are interested in developing strategies for safe shunting, then a child's train set might do. But we could also make do with a paper and pencil model with a sketch of the track and buttons representing the engines and rolling stock; or a computer model with a graphical interface and a simulator might even be more useful. Meaning (2) is appropriate.

Note that models allow complex systems to be understood, and their behaviour predicted, only within the scope of the model; they may give incorrect descriptions and predictions for situations outside the scope of their intended use. For example, a toy train set would not be much use if we were interested in the stresses and strains induced in real rolling stock when shunting. Building good models not only requires formal training, but also a lot of experience, and a critical mind.

Abstraction

> The act or process of leaving out of consideration one or more properties of a complex object so as to attend to others.

Abstraction is an important part of model building: identifying those features that are essential for inclusion in the model and separating out those features that can be neglected since the essential elements do not rely on their presence.

As an example of an abstraction, OS (Ordnance Survey) maps are used by walkers in Britain who usually want to know where they are, where they are going (how far, which direction), and how long it will take. OS maps are to scale (typically 4 cm to 1 km), and include *(easting, northing)* grid reference pairs allowing the user to pinpoint locations very accurately. For example, Perriswood near Reynoldston has the grid reference (502, 888) on the Gower map (OS Explorer Map 164). By eye or by laying a piece of string along a proposed route, experienced walkers can estimate its length fairly accurately, and then use a simple formula such as Naismith's Rule of 5 kilometres per hour plus 10 minutes for each 100 metres of uphill to estimate their walking time. To be useful to walkers, OS maps are portable (they fold flat), are to scale, and use contours and shading to show heights and indicate steepness.

The London Underground train map and A to Z street atlas have different formats from OS maps, and from each other, as they serve different purposes. The first A to Z street atlas was designed in 1936 by Phyllis Pearsall, a portrait artist, due to her frustration at getting lost during her walks through London while trying to follow an OS map. The Underground train map, on the other hand, would be of very little use to a walker. It was originally designed in 1931 by Harry Beck, a draughtsman educated in electronics, and is reminiscent of an electronic circuit board diagram with only vertical, horizontal and 45-degree lines. Train stations are not depicted geographically accurately; the connections between stations are accurate, but the stations and routes of the trains are distorted to provide an æsthetically-pleasing image. As such, it provides an ideal model for using the Tube, when you don't need to know where you are geographically as you would if you were walking, but rather are only interested in where to get on, where to change lines, and where to get off. By distorting the geography, in particular by pulling in very remote stations located at the ends of lines, a balanced and concise diagram results which is easy to use and pleasant to look at.

Notation

> Any particular system of characters, symbols, or abbreviated expressions used in art, or in science, to express briefly technical facts, quantities, etc. Especially the system of figures, letters, and signs used in arithmetic and algebra to express number, quantity, or operations.

Notation is one of the most undervalued idea in computer science. It is prevalent in the form of programming languages, but typically ignored at any higher level. A good notation provides the shortest distance between the idea in your head and a piece of paper.

Florian Cajori's two-volume masterpiece *A History of Mathematical Notations* (1928-1929) points out that scientific progress was sometimes

held back for years, decades, or even centuries because there wasn't the right notation around in which to express the relevant ideas. Compare Roman and Arabic numerals for addition, subtraction, etc. Consider also zero, the decimal point, complex numbers, the calculus. Imagine expressing sameness in quantity before Robert Recorde's invention of the equality sign. The effect that notation has on facilitating problem-solving is aptly summarised by Alfred North Whitehead as follows: *"By relieving the mind of all unnecessary work, a good notation sets it free to concentrate upon more advanced problems"*.

0.3 Specification, Implementation and Verification

The concepts and methods of computational modelling and thinking are relevant to many different fields, but their foremost domain is the development of high quality and dependable software. To set the scene, we briefly discuss three tasks that are central to software development and its formalisation through computational modelling.

1. ***Specification*** refers to the task of modelling a computing system together with its functionality and behaviour. This can be understood as a formal description of a problem to be solved.
2. ***Implementation*** refers to the task of programming the specification so that it can be executed on a computer. This can be understood as an effective solution to the problem posed by the specification.
3. ***Verification*** refers to the task of rigorously demonstrating that the implementation does indeed respect the specification. This can be understood as a proof that the implementation does indeed solve the problem specified.

The development of mathematical methods that formalise these three tasks is sometimes considered to be the *Holy Grail of Software Engineering*. In an ideal world, such methods could make software testing obsolete and software bugs history. But after four decades of research on such methods, this still remains an ideal, and there are mathematical results about decision problems, program termination and incompleteness of theories that suggest that this may be necessarily so.

However, while nobody would expect mathematical formalisation to solve all problems of science or engineering, mathematical methods and tools have significantly contributed to the success of these disciplines. The situation is similar in computing: many light-weight mathematical methods for modelling computing systems have already made their way into industrial applications from programming languages to design and analysis tools for the

specification, implementation and verification of software systems. By developing and adopting such industrial-strength methods to avoid errors, the task of searching for and repairing software bugs will hopefully become more and more unnecessary – or at least simpler and more routine – thus making "Software's Chronic Crisis" of system failures something of mere historical interest.