Russell

Introduction

Computer software has become an important component of the technological age. There are, however, long-known difficulties of developing such software – as first identified in the "software crisis" of the 1960's. The issues raised by this crisis remain unresolved and the subject of much on-going research. Despite decades of investigation and accumulated experience, developing software remains a difficult problem.

The roots of these difficulties can be traced back to the earliest days of modern software development. As processing power rose, computers were no longer confined to simple data processing tasks. Increasingly complex pieces of software were being written and the problems now associated with software development began to emerge. Computers were moving beyond tabulating census results, code-breaking and other 'number-crunching' tasks. Complex software systems such as SAGE air-defence, SABRE ticket reservation system and IBM's System/360 were forcing the development of new tools and techniques and resulting in the emergence of new kinds of software development challenges. By the 1960's the "software gap", the gulf between what was specified and what was delivered, had lead to "slipped schedules, extensive rewriting, much lost effort, large numbers of bugs, and an inflexible and unwieldy product" [NR69, p. 122]. While many of these manifestations of the software crisis persist in modern software development, some authors such as Glass have questioned to what extent we can still claim to be in crisis mode with so many examples of successful software systems [Gla98, Gla00]. Some of the issues encountered are captured in the tension between the 'theory' and the 'practice' of software development.

Typically, this dichotomy is constructed between formalised theory and informal practice. In 1969, at the second NATO conference, Dijkstra attacked the distinction labelling it "obsolete, worn out, inadequate and fruitless" and refusing to be labelled "either impractical or not theoretical" [BR70, p. 13]. Yet evidence of this distinction still exists in modern software development, with frequent appearances in book and conference titles. These two perspectives are characterised by their seemingly opposing viewpoints, each with their own staunch proponents.

The theoretical approach emphasises the structured, formalised and planned approach to software. Here development begins with a methodology – a particular approach to solving the software development problem – that tends to be followed rigorously.

Examples include structured software development methods and mathematical specification languages. However this approach is not always favoured and common criticisms include the emphasis on the formalised process that does not allow for the experimentation and unstructured development favoured by other software developers. Yet these experimental approaches to software development have been labelled as 'hacking' – where almost no look ahead planning occurs, with the development descending into a 'trial and error' approach. Agile methodologies, such as 'XP' [BA04], have evolved to provide some framework for this approach to development. While these approaches can yield results quickly and are much better placed to adapt to changing requirements, critics of this approach argue this results in an insufficient understanding of the software and the decisions involved in its development.

In terms of current practice, software development approaches tend to fit into only one of these two cultures. Few approaches can claim to fully embrace both the theoretical and practical cultures. Yet, while the divide between these two approaches might seem stark and somewhat irreconcilable, there are many reasons to be optimistic. The tensions between them are not necessarily hostile: good will exists on both sides. Efforts at bridging this divide would surely be welcomed if they genuinely made it easier to develop software. There has been

some progress on a reconciliation over the years. For example, both cultures accept that the traditional waterfall-style approaches are insufficient to account for software development practice. However, there does exist some skepticism of the possibility of a complete unification. This leads us to ask the question: why is it so difficult to bring these perspectives together? One possible explanation is that the position of theoreticians and practitioners are now firmly entrenched. Any attempt at explaining their individual perspectives must face the difficulty that each camp tends to work in very different fields. Traditionally, theoretical work is based in academic environments whereas practitioners are found in industry. Theoreticians typically see the problems identified by practitioners as further evidence of the need for a strong theoretical basis. Some reports take a pessimistic view explaining that we have "gotten away with more-or-less sloppy software development for a long time" [You98]. This view is supported by numerous headline-grabbing reports of failing IT projects. Practitioners, on the other hand, can be skeptical and usually argue that creating software will always be a difficult process that must be tackled directly. Of course these positions are far more subtle than portrayed here, and will require further exploration throughout this thesis.

The theory and practice division of software engineering culture has its limitations. It can imply an inability for theory to inform practice, or vice-versa. The techniques found in modern software engineering are not completely devoid of a theoretical underpinning nor is practice ignored in software theory. Infact, many theoretical problems are derived from issues encountered in the course of practice. Hoare has criticised this view of software engineering practice by explaining that many theoretical ideas have found their way into software practice, only that "technology transfer is extremely slow in practice, as it should be in any branch of safety critical engineering" [Hoa96]. Additionally, Hoare claims that many of these seemingly informal types of software development are deeply rooted in formal mathematical techniques and therefore showing theoretical tendencies. Equally so, theory can be informed by practice, especially in scenarios

where completely unanticipated problems occur – the identification of a potential problem in the wild can be fed back into the original specification.

For the purposes of this thesis, the author proposes to take different perspectives on the development of software. Rather than looking at theoretical and practical perspectives on software, this thesis will focus on rational and empirical perspectives. Rationalism is an "appeal to reason as a source of knowledge or justification" (TODO - reference), whereas in an empirical perspective, experience and observation are regarded as the authority to which we should appeal. This thesis will not argue that the rational/empirical divide is necessarily any better than a theory/practice divide in real terms, nor will it argue for a grand realignment of the way we view software but only that it seems more natural for the purposes of this thesis. This thesis will look at two particular sets of tools and techniques, formal methods and Empirical Modelling, and the way they subscribe to the rational and empirical viewpoints respectively.

Formal methods (FM) are a set of techniques for specifying, verifying and developing software and hardware. The techniques rely heavily on mathematical approaches for the identification and solution of design and implementation problems. A wide variety of formal methods exist, ranging from specification languages such as Z to model-checking tools such as SPIN. Due to the high-cost of use, formal methods have not found widespread use and are usually reserved for systems where high-levels of safety or security are required. Formal methods are the subject of on-going research most notably in the form of the current 'Grand Challenge in Computer Science' on dependable systems evolution. The grand challenge aims to produce general purpose tools for the verification of software systems.

Empirical Modelling (EM) is a collection of research primarily developed by Beynon and Russ [EMW] since the early 1980's. EM is an approach to computer-based modelling that is concerned with artefact construction for the purpose of sense-making activities and has been described as "thinking with computers". The tools and techniques developed at the University of Warwick have enabled

the construction of an extensive number of interactive models – all available for exploration and refinement.

This thesis will focus on an examination of the relationship between formal methods and Empirical Modelling. It will examine their individual perspectives on software development and their particular philosophical stance. Clearly, the use of formal methods to produce software is not a new topic. This thesis is not concerned with the technical details of specification languages, proof systems or refinement techniques. We ask broader questions relating to the role of formal methods in software development. We do however give attention to the software used for Empirical Modelling due to its claim of "thinking with computers" and the integral nature of the software in this claim. This thesis does touch upon some philosophical ideas but only as a consequence of the connection between the practical and philosophical ideas. A full exploration of the philosophical ideas is beyond the scope of this work.

Related Empirical Modelling work

The relationship between Empirical Modelling and Formal Methods was first explored in a thesis by Rungrattanaubol titled "A treatise on modelling with definitive scripts" [Run02]. In this work Rungrattanaubol explored how a modeller's experience is embodied in a modelling artefact. Specifically, Rungrattanaubol included a study of the heapsort algorithm consisting of an interactive model. This model was used to explore the relationship between the experiential and formal viewpoints of the sorting process.

The special-purpose EM tools, mostly notably EDEN, are discussed in-depth in a thesis by Ward [War04]. Ward gives an account of the issues to be considered when designing and implementing a dependency maintenance engine. The suitability for EM in learning environments has been explored in the theses of Roe [Roe04] and Harfield [Har08]. Further information on the underlying philosophy of EM is given by King [Kin07].

Numerous other EM publications, primarily in the form of journal and

conference papers are used throughout this thesis where appropriate. In addition to the literature referenced, a full list of published material is available from the Empirical Modelling website [EMW].

Comparisons

Before any comparison of two technologies, we must address the basis for such a comparison. As the primary focus of this thesis is an examination of the relationship between formal methods (FM) and Empirical Modelling (EM) we must briefly note some explanation for their selection.

Formal methods and Empirical Modelling both adopt different philosophical positions on software development. In simple terms, FM promotes formality, structure and planned development whereas EM promotes informality, unstructured and exploratory development. These positions seem representative of two opposing views of software and each offering radically different development approaches. Therefore any exploration of these particular technologies must also be accompanied by an complementary exploration of more general perspectives on software development. While there are other combinations that could have been selected, EM and FM seem good representatives of the issues encountered in the work conducted at the University of Warwick. Together they could be taken as representations of 'theory' and 'practice' perspectives described earlier. However, as we have already briefly noted we will examine how they fit into the 'rational' and 'empirical' perspectives.

The term 'formal methods', as used throughout this thesis, encompasses a broad range of mathematically-based verification techniques making a direct comparison more difficult. In general, formal specification languages such as Alloy, B and Z are used as representative examples in this thesis but many of the arguments presented could be extended to include other formal approaches.

It may seem unusual that formal methods, well-established in the software field, should be compared to EM, a relatively unknown set of principles and tools. Herein lies a problem that we must acknowledge: that the comparison itself is

seen as inappropriately elevating EM. The comparison is justified for two reasons. Firstly, that pre-existing published material has already addressed in-part EM's relationship to software development and the formality in modern Computer Science. Therefore, a continued exploration of these issues can only serve to clarify and further expand understanding on this front. Secondly, both formal methods and EM are taught at the University of Warwick at undergraduate level, so a fuller understanding of how they are related can only benefit students who are exposed to both of these topics.

Research Aims and Motivations

The primary aim of this research is an exploration of the relationship between Formal Methods and Empirical Modelling. Both approaches concerned are representative of particular views on software development. This thesis explores each of these topics in detail and also examines their relationship with some specific aims:

- to evolve a greater understanding of the perspectives on software development afforded by Formal Methods and Empirical Modelling;
- explore the state-of-the-art in formal software development and the perspective it adopts; and,
- use example models to explore the tools and techniques associated with both approaches.

While the thesis focuses primarily on these issues there are some additional supplementary aims that are pursued which contribute to the understanding of the EM/FM relationship. A recurring theme is educational impact of the state-of-the-art in Formal Methods and Empirical Modelling. Educational issues are intertwined with the main thesis aims due to the pedagogic nature of working with both approaches.

With the aims given above, this thesis has been guided by questions fundamental to the nature of computer software. How do programs work? How do we create programs? How do we establish the correctness of these programs? Projects such as the GC6 challenge have emphasised the legal and moral responsiblity to provide working software that reflects the increasingly pervasive nature of computers in society. It seems we have an equal responsibility to understand how developers build and shape these systems if they are expected give us guarantees about their reliability. While this thesis is driven by scientific questions about the nature of programs, it is also concerned with the ultimate impact this will have on software development practice.

The author's interest in this topic stems from observations made of the software development process and from experience gained teaching undergraduate students in modules connected to both Formal Methods and Empirical Modelling. The author is particularly interested in the role and perception of formal methods in software development. Undoubtedly much of this work has been influenced by interaction with students and guided by their attitudes towards formal methods and agile software development methods. Many students, while still new to software engineering, seem far more accepting of the difficulties of developing software and unconcerned with the questions of chronic problems. These students are far more accepting of collaborative development efforts and the agile/flexible approaches that help to manage this process. Experimentation and guided exploration seem to be activities accepted as part of interaction with computers and software systems. EM was initially conceived in the early 1980's, with some of these principles are its core. In the past twenty-five years, the author is interested to what extent these become more mainstream ideals and how the principles of EM are related to modern software development.

Research Methodology

TODO

Chapter 1

Dependable Systems, Program Verifiers and Grand Challenges

> "The general admission of the existence of the software failure in this group of responsible people is the most refreshing experience I have had in a number of years, because the admission of shortcoming is the primary condition for improvement."
>
> – E. W. Dijkstra at the 1968 NATO Software Conference

This chapter reviews some recent research in the use of formal methods for software development. Specifically, it examines the current 'Grand Challenge for Computing Research' project as established by the UK Computing Research Committee (UKCRC) for the Computer Science research community. The grand challenges are intended to be ambitious and far-reaching, in both their impact on the research community and the significance of their results. The sixth challenge, known as 'GC6', is concerned with dependable systems evolution and an examination of the aims of this challenge can help us reflect upon the state of the art in developing computer software in a formal context. One aspect of this challenge proposes the construction of a set of tools for program verification and we discuss their potential impact on both the construction of new computer programs and the analysis of existing software. However, this is not, nor is it intended to be, a

detailed critical analysis of the GC6 agenda but merely an opportunity to discuss GC6 in broad terms and examine to what extent it is indicative of the wider trends in addressing the problem of developing reliable and dependable computer software.

1.1 From Software Crisis to Program Verification

The problem of writing reliable and dependable computer programs has been a concern since the early 1960s. Throughout the 1940s and 1950s computer programming had evolved from machine code, to macro assemblers and interpreters, and then onto the first generation of optimising compilers. By the 1960s the first large-scale software projects, such as IBM's OS/360, appeared and with them a growing realisation that producing such large complex software systems was proving to be an extremely difficult task. Software projects frequently exceeded their allocated time and budget but were not able to remove all the software defects. The problems encountered by these projects are well documented, most famously in Frederick Brooks' acclaimed "The Mythical Man Month" [Bro95]. Brooks relates his experiences in managing the development of IBM's OS/360 operating system, describing how the project was plagued by slippages that resulted in time and budget overruns. As the extent of the problems became apparent, so did an appreciation of the difficulty in finding a solution. This period in the history of software development would become known as the era of the "software crisis".

The North Atlantic Treaty Organisation (NATO) conferences of the late 1960's were organised to address the problems of this "software crisis", with the first conference taking place in Garmisch, Germany in October 1968. The aim of the conference: to discuss many of the perceived problems in software and to seek out the developments that would help work towards their solution. Naur and Randell's report of the Garmisch 'software engineering' conference explained that "the phrase 'software engineering' was deliberately chosen as being provocative,

in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering" [NR69, p. 13].

The Garmisch conference marked the earliest steps in the software engineering discipline and the participants identified many of the issues that we still face today; rooted in the complex and abstract nature of software, something that has consistently proved difficult to capture. Specifically, problems identified included managing algorithmic and structural complexity, the scale of the project[1], the general lack of reliability, the nature of the design process and the lack of proper testing. Identification of these 'software engineering' problems gave rise to many different methods, processes and technologies aimed at tackling the problems. In the period following the Garmisch conference, ideas such as "structured design, formal methods and development methods" were developed [CKA04, p.181]. In the early 1970's many candidate solutions appeared to deal with the software crisis and the discipline of 'software engineering emerged. Yet no clear consensus emerged on the approach that should be taken to develop a solution to the software problem. The lack of unity and level of disagreement reflected the many different stakeholders involved in the software development process and the viewpoints of those involved with developing software. Each of the stakeholders – including, customers, analysts, programmers, project managers, documenters – had their own issues to be addressed and positions to be considered. No single approach seemed capable of addressing all of their concerns.

The techniques proposed included structured design [YC79], a top-down design technique where the system is initially described at a high-level of abstraction and then, in a series of increasingly detailed steps, moving towards the implementation level. Such development models were viewed "as much a management tool as a technical one" [CKA04, p.182] where the entire software life-cycle was recognised, considered and managed. Techniques such as structured

[1]Given as the "number of different, non-identical situations which the software must fit" [NR69, p.68].

design were in stark contrast to that of formal verification, the process of checking whether a program matched its specification, another technique that emerged in the early years of software engineering. Formal verification was designed to reason formally about a program and its specification in order to produce a proof of correctness for the specified properties.

The desire to find a solution to the software crisis was strong and new developments were accompanied with a rhetoric of a software panacea. New techniques brought the promise of breakthroughs but ultimately failed to produce dramatic progress. Formal methods seems particularly badly affected by the sweeping claims and generalisations that have plagued new developments in software engineering[2]. This has resulted in a negative view of formal methods that is widespread and one that is difficult to repair. Unfortunately any solution to the software crisis would not prove to be as neat as finding one all-encompassing technique or approach. While the desire to find a single general solution was strong, balancing the needs of the many stakeholders proved to be a tricky endeavour. Despite all the solutions explored, as Shapiro in a review of early software engineering [Sha97], notes:

> "No solution aimed at a single area could provide the degree of relief many were seeking. Moreover, agreeing on singular approaches with respect to any of these issues also frequently proved difficult in the face of incommensurable philosophies and inescapable trade-offs".

This encapsulates the central tenet in Brooks' "No Silver Bullet" paper, in which he states that there will be no single achievement that "by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity" [Bro87]. Unfortunately many of the problems identified at the time of the software crisis have persisted to the present day and few would claim that any solutions have been fully realised. While today's software problems are not of the scale or frequency of the 1960s, especially relative to the size of modern software

[2]The criticisms levelled at formal methods are explored in chapter 2.

projects, they do still exist and the fundamental problem remains. At the time of the early NATO conferences, computers designed for specific purposes were being replaced by general purpose computers. Where hardware once dominated, software was beginning to be recognised as the interchangeable and malleable component of a computer system. And while the speed of development in computer hardware has been rapid, with transistor density doubling approximately every two years[3], the progress in computer software has been much more troubled. Programmers struggled with issues that did not exist in computer hardware – they were unable to benefit from the regularity and repeated patterns found in hardware.

Following the first software-engineering conference in Garmisch, a second conference titled 'Software Engineering Techniques' was organised a year later in Rome, Italy. This conference was to focus more on the technical details with far less emphasis on the organisational and management issues that featured at the first conference. In the conference report[4] [BR70], the editors Buxton and Randell noted that:

> "The resulting conference bore little resemblance to its predecessor. The sense of urgency in the face of common problems was not as apparent as at Garmisch. Instead, a lack of communication between different sections of the participants became, in the editors' opinion at least, a dominant feature. Eventually, the seriousness of this communication gap, and the realization that it was but a reflection of the situation in the real world, caused the gap itself to become a major topic of discussion".

In [Sha97], Shapiro discusses the outcome of the 1969 conference and concludes that this perceived gap was "generally regarded as one between theory and prac-

[3]This trend in the development of hardware is known as Moore's law, first observed by Intel co-founder Gordon Moore [Moo65]

[4]Brian Randell relates his experiences of this conference and of writing the report in an Annals of the History of Computing article titled "Memories of the NATO Software Engineering Conferences" [Ran98]

tice". A discussion, from the final day of the conference, on this topic is featured at the very beginning of the conference report — a sign of how much the topic troubled many of the attendees. Nowhere was this gap more apparent than in the use of formal methods, one of the proposed solutions to emerge from the crisis.

1.1.1 Verifying Compilers

One of the ideas in formal methods was that of a verifying compiler, first proposed by Floyd in 1967 [Flo67]. A verifying compiler is a software tool that, given both the program and a specification for that program, is able to determine correctness – whether the program does indeed follow the specification – through a process of formal reasoning. In the following year, Dijkstra proposed that assertions should be written before the program itself. He argued that the program would be derived from these statements – resulting in a program annotated by a series of logical assertions. Dijkstra hoped that this would be "a relevant step in the process of transforming the Art of Programming into the Science of Programming" [Dij68]. Unfortunately, given the limitations of theorem provers and the inaccessible nature of most software code, progress along this path stalled preventing the development of such a verifying compiler. The mathematical theory of programming was still in its infancy and the capacity of computers was a fraction of their modern day equivalent. Forty years since Floyd's paper the problem of program correctness still remains, as does the attractive idea of building a verifying compiler.

1.2 Grand Challenges in Computing Research

Grand challenges are designed to be projects aimed at difficult to solve problems – "directed towards a revolutionary advance, rather than evolutionary improvement" [GCW08]. These projects typically require a large-scale collaborative effort because any solution is expected to be beyond the capabilities and resources of a single person or team. While efforts of this magnitude take place over an ex-

tended period of time, usually measured in decades, the projects can help to focus and to organise a research community's output. The challenges seek to advance scientific understanding by setting out a common goal, decided by the research community, towards which many researchers can direct their efforts. Although the exact steps the research will take are not yet known, the ultimate goal is clear enough to direct researchers along the initial path. And although the project is not guaranteed to solve the original problem, it is hoped by the very process of investigating it that there will be a significant advancement in the theoretical understanding and many practical applications will result. There are many historical examples where scientific research has produced some quite unexpected and unrelated discoveries, e.g. Röntgen's discovery of the X-Ray, Fleming's bacteria experiments leading to the discovery of Penicillin and Pasteur's immunisation methods. None of these discoveries were predicted at the outset and were not the result of goal-directed research. The results are attributed to the serendipitous nature of scientific research. The grand challenges hope to spur innovation in pursuit of its goal. With specific reference to Computer Science, Hoare notes that "the primary purpose of the formulation and promulgation of a grand challenge is the advancement of science or engineering" [Hoa03b]. A grand challenge can only be developed when it is believed the current research path is sufficiently mature. Research within a field must have advanced to the point where it allows for prediction and planning on a long-term scale. There are many notable examples of 'grand challenges' outside the computer science discipline which have resulted in large-scale advances. Examples include the space race between Russia and the USA in the 1960s and the human genome project in the 1990s.

The "Grand Challenges In Computing Research", as established by the UK Computing Research Committee (UKCRC), hopes to make similarly significant contributions to the development of research in the discipline. A programme of seven computer science grand challenges has been in development, each with a different research theme [GCW08]. These challenges are expected to be long-term research goals for the computer science community requiring significant invest-

ment and commitment. The success of these challenges is not assured and this is acknowledged by the UKCRC at the outset. Each of the seven themes represents a separate grand challenge ranging from the further development of quantum computing, to distributed computing to dependable systems. Details of each of the challenges can be found in [HM04]. This thesis is primarily concerned with the sixth challenge, titled 'Dependable Systems Evolution'. Hoare is careful to establish this Grand Challenge as pure scientific research with the aim of answering some of fundamental questions concerning the nature of programs [Hoa07b]. The development of any grand challenge shows that a consensus has been reached within the community on the future research directions. Further, it shows that the subject matter has reached a level of maturity sufficient to make detailed judgements on potential developments. While the exact final outcome of this research is unclear, researchers have agreed that it is a worthwhile investment of time and effort likely to yield many interesting research opportunities.

Full details of the grand challenge criteria and the dependable systems project's suitability for such a challenge are given in [Woo03].

1.3 Dependable Systems Evolution – GC6

The introduction of computers into everyday life continues to expand and we increasingly depend upon them to perform everyday tasks, sometimes without us even being directly aware of their existence. Although many of these systems perform routine functions, there are numerous other systems that we entrust with safety-critical tasks such as medical radiation therapy, nuclear reactor control systems, railway signalling systems and air traffic control systems. Yet the unreliability of computer systems has become an accepted part of daily life, with many well-documented failures of computer systems ranging from space probes such as Mars Climate Orbiter [EJC01] to medical systems such as the Therac-25 radiation machine [LT93]. While most of the computer problems we encounter are only an inconvenience, there are an increasing number of these safety-critical

tasks performed by computer systems upon which lives do depend. Software engineers must therefore be capable of building systems that are able to meet the highest levels of reliability. The capability and knowledge of how to consistently build reliable and dependable computer systems remains elusive but the attempt to address this problem is the cause taken up by the GC6 "Grand Challenge" on dependable systems.

Developing software is an inherently difficult task due to the complexities of making sense of the subtle system details. The challenge begins at the earliest stages of the development process. Software developers frequently note the difficulties encountered when trying to write requirements specification documents for a new piece of software. The requirements capture process directly challenges our knowledge, understanding and ability to make sense of a system, and also our ability to translate this knowledge precisely and unambiguously into written form. In the later stages, at specification and design of the system, the possible modes of interaction, desired features, the mechanics of the system and the nature of interactions with other computer systems must be anticipated. No system can be regarded as a closed or static entity because it must be able to respond to requests to add or revise functionality and incorporate these changes into our original design. As the understanding of the original problem evolves, a developer must be prepared to alter the software accordingly. Further exploration of the software can bring new understanding or a realisation of changes that are needed and a development environment must allow these new ideas to be incorporated. There are, of course, the inevitable programming errors, for example: mistakes in syntax, typographical errors and badly constructed solutions. Whatever problems are encountered, the core of the problem lies in the ability to know what system to build and then how to build it well.

In response to this agenda, as part of the "Grand Challenges in Computing Research" initiative, 'GC6' was formed, a project primarily concerned with 'dependable system evolution'. That is, to learn how to build systems that can be developed to be reliable as possible with as few defects as can possibly be

achieved. As described earlier, the grand challenges are designed to help focus attention on a long-term research goal. As part of these challenges, GC6 is expected to be a long-term project[5] which will be forced to address many different kinds of problems but ultimately addressing the growing need for reliable computing systems. Guaranteed software reliability, one of the stated aims, would allow reliable predictions to be made about the dependability of the software and how it will perform under given set of conditions. Woodcock [Woo06] asks whether software will be issued with warranties that make guarantees about their reliability[6] and he believes that the application of formal methods is the cheapest way to achieve this, noting that "the use of formal methods will become widespread, transforming the practice of software engineering". As explained in one of the original GC6 proposals [Woo03], defining the nature of a dependable system:

> "A computing system is dependable if reliance can *justifiably* be placed on the service that it delivers, characterised in terms such as functionality, availability, safety and security. Evidence is needed in advance to back up any manufacturer's promises about a product's future service, and this evidence must be scientifically rigorous."

To make this level of guarantee about the reliability of the system, evidence must be gathered to support the claims and in a form convincing enough to justify the guarantees. As the system evolves through development, changes will occur that could affect the basis of any reliability claims. Rapidly evolving systems are increasingly commonplace in Internet and e-business environments and any work on dependable systems must be able to cope accordingly. The ability to verify a system at all stages of development is highly desirable. We need to be capable of generating this evidence. There are two possible means of generating this evidence: software testing, and formal mathematical proofs.

1. **Software testing.** Testing is the form of evidence gathering most fre-

[5] According to [Hoa07a], approximately a billion dollars across 10 years.

[6] Woodcock is, in part, relating the story of Tony Scott, the General Motors' chief technology officer, who asked these questions in an interview with *eWeek* [Dig04].

quently used in modern software engineering. Software is subjected to a battery of experiments designed to uncover faults in the functionality, reliability, safety or security properties. Discovering defects relies upon a certain degree of happenstance: that the chosen input data and sequence of actions reveal the flaws in a readily apparent way. We cannot be guaranteed to find the defects that exist. However diligent our testing procedures they are not exhaustive – there is always the possibility of a sequence of inputs or a scenario for the system that we have not conceived, which will subvert the expected behaviour of the system. Development methodologies such as XP [BA04] have seen the proliferation of techniques such as unit testing, where individual components of the system are tested automatically and repeatedly by a software tool.

2. **Formal verification.** Formal proofs, the method advocated by GC6, is the second option and is the process of developing a formal mathematical proof for the correctness of the system. In this instance, a formal proof might take the form of a series of mathematical derivations that logically show how and why a program meets a specification of the system. The specification would take the form of a higher-level abstract description of the system behaviour. The formal verification step could take the form of either theorem proving or some form of model checking. Unlike testing, a formal proof could be exhaustive and it would be possible to, with respect to a specification, prove that a program is correct. The advantage of primarily automatic testing is that it could be rerun many times without significant human intervention.

Both of these methods provide means for the discovery of faults within a system, with the aim of improving the reliability of the system. Unfortunately the cost of producing such evidence can be prohibitively high. Unit testing, for example, increases the initial cost of development dramatically because of the time and money required to specify and code the tests. The business perspective perceives

development time to be 'wasted' on writing unit tests for code that already works. While writing unit testing requires no particular skill set beyond those required to develop the software itself, formal specification does require a specialised skill level and a higher degree of proficiency in mathematical reasoning. It takes many hours to devise and execute tests on a software system, and mathematical proofs about the properties of a system are produced at even greater expense and take longer yet. A workforce with the required skill set inevitably costs more money, further increasing costs of the development. The GC6 project aims to reduce the cost of these mathematical proofs so they are a more accessible and practical option to test a computer system. Hoare notes that:

> "Critical applications will always be specified completely, and their total correctness will be checked by machine. In many specialised applications, large parts of a program will be generated automatically from the specification." [HM05]

Any process of complete formal checking of software requires several components. There must be a sufficiently descriptive formal specification language. That is, either one single notation, or several notations in a valid combination, capable of fully describing the system we wish to build. There must also be a language which describes the implementation of the system. And, we must have some kind of transparent[7] mechanism which is capable of comparing the specification and the implementation. Currently, formal methods only allow us to examine the specification and do not yet have sufficient scope to encompass program code checking (the aim of GC6). The process of discovering and correcting faults improves the correctness of the system with respect to a specification. As noted earlier, the dependability of a software is described in terms of its reliability, safety, availability and security. We can incorporate checks for various safety (e.g. deadlock) and security properties into our formal proofs. It is sometimes difficult to act with foresight in these matters.

[7]Tools and techniques that do not allow examination of the mechanism (e.g. proof rules) would not seem appropriate.

It can be difficult to envisage the types of problems our system might encounter before we can even attempt to express these properties and check their validity. The correctness of a system is concerned with the adherence of the program to the specification. Any measure of dependability is reliant upon the ability to perform the appropriate tests on the program and specification. Unless the right type of properties are discovered, we cannot be sure our system is dependable. And even then, we cannot know if there is still some safety or security property we have not considered or for which our tools are unable to check. There is a measure of happenstance required to provide evidence in this situation. Properties such as availability are difficult to quantify, and there is definite scope outside the correctness of the software. Any measure of reliability must take into account the underlying operating system (if any), situational aspects of the system, and the hardware on which the software is executing. Safety and security are the easier properties to quantify as it can be checked if there is any possible input to the system that will lead to an unsafe or insecure state respectively. So a careful distinction must be made between correctness and dependability, and that a correct program with respect to its specification does not necessarily imply that it will be dependable.

1.4 Verifying Software Systems

1.4.1 Tools for Program Verification

One of the barriers to the adoption of formal methods within the software engineering community has been the lack of tool support. The proponents of the GC6 project identified this problem early and noted that the "realisation of our vision will depend on the development of a powerful set of tools for strong software engineering" [Woo03]. It is clear that widespread adoption of the research will be difficult otherwise but what remains unclear is exactly what form the tool should take. Early suggestions were numerous and wide-ranging. The choice of tool would have a considerable impact on the success of the project. While some

formal methods tools already existed[8], the vision for the GC6 endeavour would require a tool that would support the revolutionary, rather than evolutionary, advancements.

Tools suggested in one of the early GC6 proposals [Woo03] included tools to construct specifications from various system properties, tools to generate systems from pre-existing components, verifying compilers, invariant generators to extract specifications from existing code, refinement calculus support tools and tools to perform automated testing in various forms. Ultimately, the project coalesced around a set of tools for program verification (or program verifiers[9]) and these are now a major component of the response to the dependable systems grand challenge [Hoa03b, HM05]. A program verifier accepts two inputs, a specification and a program, and attempts to show that the program correctly meets the specification. This allows guarantees to be made for the behaviour of a program – to prove the correctness of a program with respect to its specification.

There are examples where formal methods have been successfully applied to a specific problem domain. For example, Microsoft Research's SLAM project[10] has applied static verification techniques to the problem of checking for errors in device drivers. It is claimed that unreliable device drivers cause "85% of recently reported failures" in Microsoft's Windows XP operating system [SBL03]. Bill Gates, in a 2002 keynote address at Windows Hardware Engineering Conference, said:

> "Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability."

[8] An exploration of existing formal methods tools appears in chapter 2.

[9] It should be noted that the term 'verifying compiler' is the original name for what was then relabelled the 'program verifier', which is now in turn referred to as the 'tools for program verification'. Consequently the terms 'verifying compiler' and 'program verifier' appear frequently in much of the literature surrounding the grand challenge.

[10] `http://research.microsoft.com/slam/`

The SLAM project has developed a toolset that can be used to verify the temporal safety properties of C programs [BR01]. By reducing C programs to boolean programs (where all variables have the boolean type), various safety properties can be automatically checked. The project has applied this technique to driver verification with the Static Driver Verifier (SDV) tool [BBC+06]. The SDV tool models the kernel environment and is able to check how the driver will behave according to a set of API rules. These rules describe the expected behaviour, e.g. correctly acquiring and subsequently releasing a lock, interacting with the plug-and-play mechanism, and dealing with power management correctly. For example in one particular test "running SDV on 126 WDM [11] drivers with over 60 rules" found "multiple errors in almost every driver" [BBC+06]. Clearly, this tool has been successful in identifying potential problems and may be used by many device driver developers not just those with detailed knowledge of formal verification methods. Herein lies the success to this tool – it is not necessary for the user to be exposed to the underlying formal techniques. Device driver developers have not been required to perform the specification step – this is already preprogrammed into the SDV tool – nor do they need to annotate their driver code – the toolset does this automatically [BR02]. The tool itself embodies the knowledge of the experienced kernel developers, of what problems might occur and how to identify them. This set of knowledge has been built up over many years of study of the operating system kernel and problematic device drivers, and is something we could not expect from any individual developer simply from ad-hoc testing of their driver. The success of the tool is closely bound to the domain knowledge of the SDV developers.

1.4.2 Establishing a frame of reference

The GC6 projects aim to produce a far more general purpose tool, where the specifications would need to be provided and automatic invariant extraction might

[11] Windows Driver Model – the driver model used in Microsoft's Windows 98, 2000, XP and Vista operating systems.

not always be possible or appropriate. As we shall explore in Chapter 2, there are several general purpose formal development tools that are available, for example, the B-Toolkit which allows abstract specifications to be incrementally refined to an implementation level before translation into a high-level language such as C. While formal languages such as B have proven useful in some problem domains, the limitations quickly become apparent when compared to high-level languages commonly used for software engineering projects. Programmers who work with unfamiliar formal specification languages can find it a highly discouraging experience. Lack of familiarity with formal approaches and inadequate mathematical skills sets the barrier to entry much higher than with programming languages such as Java. The specification languages that are supported by this class of formal methods tools are quite different from the programming languages a developer might expect to use. Program verifiers aim to expand the scope to include a high-level language that mainstream software developers would find more comfortable. In fact, it is hoped that the corpus of open source software, written in languages such as C, C++ and Java, would be used as a means of testing the program verifier.

The correctness of a program cannot be verified without an appropriate frame of reference. This would take the form of a specification describing the expected behaviour of a program. These specifications could appear as annotated program code, where assertions about the program are attached to various sections of the program code. Annotated program code is a style of programming similar to Knuth's literal programming [Knu92] but differs in that the description takes a purely mathematical form to allow them to be read by an automated verification program. Assertions within the program code itself will be disregarded by a traditional compiler and would not interfere with compilation. A verification tool, however, would be able to make use of these assertions, which make definite statements about the properties of the program. The assertions would be evaluated with respect to the program code and allow a determination to be made about the correctness of the program. We cannot hope for absolute correctness

but the aim is to make the program as correct as reasonably possible.

Hoare suggests that upon completion of the verifying compiler it will be possible to perform post facto verification on some of the plethora of open-source code available and ready to be used as a test-bed for the verification tools [Hoa03b]. It is expected that this existing open-source code will be examined and then annotated with assertions. This software would then be allowed to enter a library of verified software components. Whether this code is built with formal tools or verified through a post-facto checking process, the components in this library would have been proved as correct with respect to its specification. Raising our confidence in a library of reusable components could lead to a shorter development cycle – specifically between implementation and delivery. If achieved, extensive testing would not be obligatory before deployment reducing the cost and easing the release schedule. Advancing the tools to a level that could achieve this is expected to require a considerable effort and will take the project at least fifteen years [Woo03].

1.4.3 Developing formal tools

Formal verification tools are considered one of the major components of the dependable systems "grand challenge". The high-degree of automation provided by these formal methods tools is necessary in order to manage the larger specifications and proofs. This automation is one of the biggest advantages over pencil-and-paper efforts. Of course it would still be possible to manually perform any of these verification efforts but the resources required would likely make this impractical. So the benefits of a tool capable of automatic checking on a large scale should not be underestimated. The use of these tools a degree of trust in their integrity. The verification results given to a developer by a tool will have a large impact on the extent to which they will trust their specification. Therefore, a developer should always accept the results given by the verification tools with a healthy degree of skepticism. So it seems wise to consider the method by which the verification tool itself will be constructed.

A developer would wish any program verifier to have the same properties as the programs we are aiming to construct - reliable, dependable and bug-free. Yet, clearly the program verifier cannot be checked with the very tool under construction and nor would it increase the accuracy of the tool. The original GC6 proposals acknowledge this problem. In [Woo03] it is noted that the program verifier[12] does not have to be verified:

> "Note that the verifying compiler itself does not have to be verified. It is adequate to rely on the normal engineering judgement that errors in a user program are unlikely to be compensated by errors in the compiler. Verification of a verifying compiler is a specialized task, forming a suitable topic for a separate grand challenge."

The research community has resolved to build the verification tool by the usual software engineering methodology, and relying upon competitive testing against competing verification tools. In "The Verifying Compiler: A Grand Challenge for Computing Research" [Hoa03b], Hoare whilst discussing grand challenge criteria[13], says that the verifying compiler will have been tested with "millions of lines of open source software" and that the "proofs themselves will be subject to confirmation or refutation by rival proof tools". In a talk at the Grand Challenges of Informatics Symposium in Budapest [Hoa06], Hoare further explained that "as engineers we can trust checkers because they give an independent assessment of the correctness which is backed up by testing and experience; another tool in the armory". Without resolving the problem completely this represents a pragmatic solution to the problem but does start to expose some of the difficulties when developing such a tool.

There is a potential solution to the 'verifying the tool' problem that attempts to minimise the potential risk - develop a bootstrapping program verification tool. That is, to construct as simple a verification tool as possible, and

[12]It is still referred to as the 'verifying compiler' is this paper.

[13]These are the criteria, identified by James Gray in [Gra03], to use when deciding if a project is a "grand challenge".

then use this to verify another more fully-featured verification tool. In a series of increasingly elaborate development steps a tool with the required functionality is developed. What remains an open question is exactly what level of functionality is required at the initial step to develop a successful end product. We still have to construct the initial step by hand, employing careful manual checking of the first version of the tool. There will always remain some level of uncertainty with lingering questions.

Does the compiler produce correct machine code? Does the hardware execute this machine code as expected? Will there be any unexpected change in conditions which might affect the system?[14] There is no easy solution to this problem, and careful management of effort is needed to address potential problems.

A different approach would be to forget automatic verification with a program verifier and resort to a human manually performing the necessary calculations to check the correctness of the program. Over a large proof, the speed and accuracy of a human is unlikely to match an automatic verifier. A manual verifier would also be prone to mistakes, especially as the size of the program requiring verification increased. Clearly, producing a workable tool has some distinct advantages. The tool can be tested on a variety of programs, helping to identify any potential problems. Also, modifications can be accounted for quickly and easily - something that is difficult with a manual process.

1.4.4 Correctness, assertions and specifications

A method for ascertaining the correctness of a program and whether it exhibits appropriate behaviour is required. A prerequisite for this assessment is some basis for comparison – a statement of intent for the program. A specification for a program, written at a higher level of abstraction, is exactly that. The specification is a description of *what* must be true in a program but does not specify exactly

[14]For example, consider processors which must work in space satellites and other environments where high levels of ionising radiation are present. To guard against errors, such as random bit-flips, processors must be specially built on insulating substrates such as sapphire.

how it should be achieved. Any evaluation of program correctness must be given *with respect to* the program's specification – any other determination failing this requirement should be treated with suspicion and a degree of incredulity. When such importance is placed on a specification it necessitates one that is of good quality and fit for purpose.

One technique for the specification of a program is the use of assertions. Logical assertions are boolean statements - predicates used to state a premise. We consider these premises as factual statements about a program and they can serve as annotations of the existing program code. Therefore, assertions added to program code are able to serve as a kind of post-hoc specification for a program. At the critical points in the program code we we assert the state of the program at that point in its execution, allowing a clear demonstration of correctness. If the assertion fails, a problem has been identified with respect to the stated specification and the correctness of the program is in question. There are no limits to the scope of assertions and they may range from simple properties (e.g. that an addition is performed correctly) through to complicated and wide-ranging functional and non-functional properties.

Many programming languages have support for assertions, yet the level of this support varies quite notably. Run-time assertion checking (RAC) allows assertion statements to be checked during the execution of a program by including boolean statements within the original program code. The Eiffel programming language has "built-in" support for the design-by-contract method [Mey92a]. Languages such as C and C++ have the `assert` macro, used at run time to provide some degree of type checking [RK88]. This macro simply expands into an in-line `if` statement that is capable of aborting the program in the false branch. A similar `assert` construct is available in Sun's Java programming language [GJSB05] but does not work by macro expansion to an in-line statement. The Eiffel language has far richer support, including several assertion possibilities such as `check`, `require`, `ensure` and `invariant` [Mey92b]. Tools exist to augment the assertion functionality of existing languages, such as the Annotation

PreProcessor (APP) described by Rosenblum. In relating the experiences with the APP, Rosenblum also makes efforts to characterise the kinds of assertions that were most useful in discovering faults in program code [Ros95].

Despite the usefulness of assertions in evaluating the correctness of a program, the difficulty still remains writing these assertions. Constructing assertions that are succinct, accurate and meaningful can be tricky. Adding them to code can be costly in terms of time and effort. In [Hoa03a], while reflecting on his own involvement in the use of assertions, Hoare addresses this problem:

> "A common objection to Floyd's method of program proving was the need to supply additional assertions at intermediate points in the program. It is difficult to look at an existing program and guess what these assertions should be. I thought this was an entirely mistaken objection. It wasn't sensible to try to prove the correctness of existing programs, partly because they were mostly going to be incorrect anyway. I followed Dijkstra's constructive approach to the task of programming."

Here, Hoare warns against the verification of existing programs because, due to their construction through informal means, they are likely to be incorrect. It is, however, still hoped that this approach will be used to test the program verifying tools. Hoare explains that the tools will "have been tested in verification of structural integrity and security and other desirable properties of millions of lines of open source software, and in more substantial verification of critical parts of it" [Hoa03b]. Clearly, there is the desire to perform some kind of post-hoc verification on existing software but the difficulties may deter many software testers. Dijkstra was also critical of this approach [NR69, p.31], explaining that:

> "I am convinced that the quality of the product can never be established afterwards. Whether the correctness of a piece of software can be guaranteed or not depends greatly on the structure of the thing made. This means that the ability to convince users, or yourself, that

> the product is good, is closely intertwined with the design process itself."

Nevertheless, some software companies such as Microsoft have found success with this approach where "assertions figure strongly [and a] recent count discovered more than a quarter million of them in the code of its Office product suite" [Hoa03a]. Regardless of whether a correctness by construction approach is favoured or not, it may simply not be possible to adopt this approach because of financial or time constraints. Developers rarely have the resources required to start afresh on development, so a post-hoc verification may be the only viable option in many cases.

Any kind of post-hoc verification relies on the ability of a developer to extract meaning from code and synthesise this into a formal model. Jones, in [JJJ+07], relates his difficulties with this approach, noting that:

> "I have several times been invited to use formalism to construct a post-factum verification of an extant program. I confess that I have always had to abandon the undertaking and design a new program from what I understood to be its specification. Only in this way can one find the abstractions which make verification tenable."

There are two problems that exist here. The first is the difficulty associated with writing assertions and the second is a developer's ability to understand a program sufficiently in order to produce a formal model from which it can be verified. Even if we have access to documentation, the problem of program comprehension is a difficult one. Programs that consist of thousands of lines of code can be very difficult to understand. Without detailed knowledge of the code to be analysed and verified, it is difficult to find valid abstractions and make decisions on how a specification should be structured.

In any case, the reality is that supporting documentation is likely to be inaccurate, incomplete or simply non-existent. Even with access to the original system developers, it may be difficult to expose subtle details of the system -

long forgotten in a myriad of other projects and design decisions. The development history of systems are littered with small changes and tweaks made to elicit the correct behaviour. These changes further skew the ability to understand a system as differences from the documentation confuse, or worse, make it appear something works in a particular way when in fact it does not due to some subtle change. It becomes clear that understanding cannot come simply from an examination of the program code but must come from experiment of both the specific program and the behavioural properties of the programming language constructs. Another possibility for examining the behaviour of a system is a targeted testing of the code. Program tests, or 'unit tests', are a common component of the iterative programming software development methodologies such as 'Extreme Programming'[15] [BA04]. These tests are an alternative approach to assertions and take the form of small test cases that are frequently tested against the system. The experimental nature of these tests helps perform the verification (but not in a formal sense) for software written in the XP style. Tests are commonly created upon the discovery of a defect to ensure that if the error is ever accidentally reintroduced into the code it will be possible to discover it. The reason for the success of tests over assertions is due to a combination of factors, some technical and some practical.

Any work on a program verifier must be accompanied by a equivalent amount of effort understanding the processes of comprehending program with the aim of writing assertions to annotate existing program code.

1.5 Rationality and GC6

The GC6 grand challenge is clearly an ambitious project requiring many significant contributions in terms of research output. Undoubtedly, such ambition can serve the project well and help further the state of the art in formal software specification and development. To what extent this agenda serves the wider problem

[15] This is frequently abbreviated to just 'XP'.

of thinking and developing with computers is, as yet, unaddresed. The purpose of this discussion, and indeed this chapter, is to be skeptical but not pessimistic about the future role of formal methods in software development. If we take a closer look at formal methods, it is possible to ask some probing questions of its past record and possibilities for its future. One such question is why have formal methods not succeeded in the wider software development community? Possible explanations include the unfamiliar notations, developers' lack of mathematical background or the lack of good tools. Any answer is connected to all of these issues but the details seem to lie in the underlying approach formal methods takes to software development. This section introduces the concepts and rational and empirical views of software development and explores to what extent formal methods endorses the former.

With a similar focus on answering fundamental scientific questions, some have questioned the impact of FM on software engineering research, e.g. the responses of Jackson, Holcombe and Tully to Hoare in [JJJ+07]. This underscores one of the difficulties of such a grand challenge - we can not foresee the long-term impact, beyond the pure scientific research itself. Hoare has previously addressed this concern, stating that "modern software engineering practice owes a great deal to the theoretical concepts and ideals of early research in the subject" but accepts that it can take a long time to realise these results [Hoa96]. The project envisions that, in 20 to 50 years, programmers will make "no more mistakes than professionals in other disciplines" and that "their remaining mistakes will be detected immediately and automatically" [HM05].

Developing software is, however, not just concerned with producing code, it is a complicated activity which must also involve designers, users and domain experts. Software systems cannot exist only in a formal world, they must exist 'in the wild' where phenomena are unreliable, interactions with users are unpredictable and the ability of other systems to respond appropriately is uncertain. Software is also increasingly subject to changing requirements and must be able to change and evolve appropriately. Programming techniques that do not account

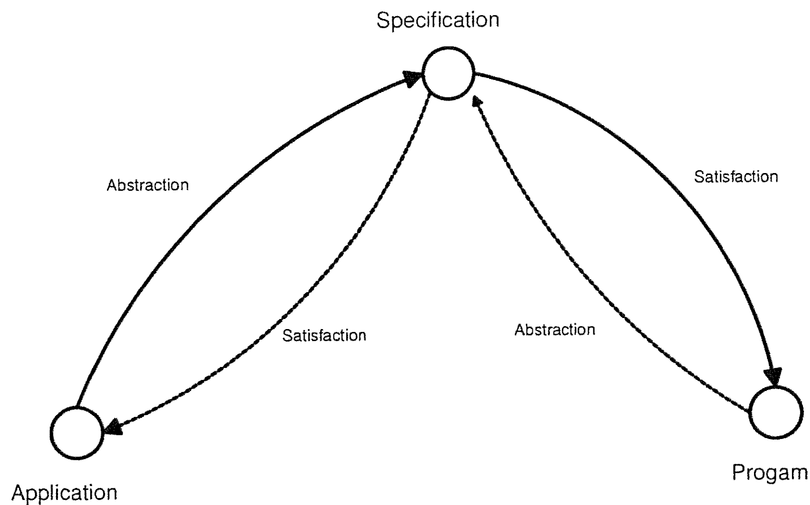

Figure 1.1: Relations between specification, program and application *(recreated from [TM87, p.11])*

for this style of software development seem to be at an inherent disadvantage. Aside from the technical aspects of software development there are also many social and organisational difficulties. Holcombe notes that "high profile IT failures are caused – not by inadequate technical design – but by political, organizational and human failures" [JJJ+07]. Large software projects seem particularly troublesome with Yourdon in 1998 claiming that "approximately 25 percent of large, complex projects are cancelled before completion" [You98]. In 2005, Charette, citing several large project failures, claimed that of "the IT projects that are initiated, from 5 to 15 percent will be abandoned before or shortly after delivery as hopelessly inadequate" [Cha05].

It is important to consider the potential sources of error and how the different techniques can be applied to these problems. Figure 1.1 shows a diagram, from Turski and Maibaum [TM87, p.11], that illustrates the relationships between application, specification and program. The diagram shows the satisfaction relation – the relation indicating whether a program satisfies the specification. They also consider an 'abstraction' relation between application and specification – whether a specification is a valid abstraction of the application. Both these relations are a potential source of error and, as shown, the success of the specification/program

relationship is dependent on the success of the specification/application relationship. Yet the relations are separate, one is concerned with the program satisfying its specification, and the other is concerned with a specification accurately describing the problem. Jackson terms the specification a "logical firewall" as it divides the concerns of " 'building the program right' and 'building the right program' " [Jac06b]. Consequently, it is possible for the specification/program relationship to be satisfied but fail to produce the desired effects due to an inaccurate model of the world. All of the formality (and the optimisation) exists between the program and specification and it is this relationship which can be formally reasoned about. With such great risk at the early stages of the development process, care must be taken to ensure the requirements are gathered accurately and that the initial specification is correct. The impact of a mistake introduced at an early stage can be magnified greatly at a later implementation stage. Finding these mistakes can require an intensive effort but is necessary to reduce the likelihood of them occurring. Hoare acknowledges the importance of these early stages, noting that "the discovery at this stage of only a single error and a single simplification would fully replay all the effort expended" [Hoa96].

Another area of consideration is how a specification is written. How is the initial specification produced? The relationship between the problem and the specification cannot be formalised, is not calculable and so we cannot apply verification techniques. Checking the specification must rely on a process of validation to check whether the specification is an accurate representation of the problem. There is no more authoritative document against which the specification can be judged – it is the point at which the formal and informal worlds meet. The informality of the problem domain must be constrained sufficiently, yet accurately, in order to capture the correct details in the specification. Hoare, relating his experience with assertions in [Hoa03a], expressed his views on the character and form a specification should take:

> "There is no conceivable way to prove a specification correct – against what specification would that be? Such a higher-level specification if

> it existed, should have been chosen originally as the starting place for the design. So the only hope is for developers to make the original specification so clear and so easily understandable that it obviously describes what is wanted, and not some other thing. That's why it would be dangerous to recommend for specification anything less than the full language of mathematics. Even if this view is impractical, it represents the kind of extreme in expressive power that makes it an appropriate topic for academic research. Certainly, if the basic mathematical concepts turn out to be inadequate to describe what is wanted, there is little hope for help from mathematics in making correct programs"

It seems clear that more research is needed in this area. Jackson subscribes to this view, calling for further developments in the way software is described and attention paid to the education of new software developers:

> "In our pursuit of well-engineered descriptions of the real world we should recognize – and every student of software engineering should be taught – that formalization can show the presence of description errors, but not their absence." [Jac98]

Cliff Jones, in [JJJ+07], takes a similar view, reflecting that the problems occur when the specification is not always developed with a true sense of the environment in which it will be expected to operate:

> "The most common cause of failure in a system is not that the program fails to match its specification but the fact that the specification of the technical component makes no sense when combined with the expectations, professional notions of responsibility and confidentiality of the human components of the system."

The discovery of these errors still relies on our ability to conceive of appropriate error scenarios and the potential consequences for our software. The

problem of unknown and unexpected external forces is well-documented in engineering and construction. Henry Petroski's idea that was not the result of a failure to check the design but that "the possibility of failure of the Tacoma Narrows Bridge in a crosswind of forty or so miles per hour was completely unforeseen by its designers, and therefore that situation was not analyzed." [Pet92, p.165].

Nevertheless, writing the initial specification is a difficult problem and one that may be expected to remain so. Achieving a fault-free system is not an easily achievable goal and the problems might not be solely down to problem with the specification. There is also the question of to what extent we need correctness and whether other attributes are more important. Turski sees a softening of attitudes with correctness being replaced by dependability resulting in "a deep shift in the perception of quality" [Tur03]. Turksi emphasises that building a system to catch all the error cases is not viable and that sometimes recovery is a better option. Effort spent on an appropriate recovery mechanism can be more beneficial. "Crash-only" software embraces this view of software development – that software should be able to crash safely and then recover quickly [CF03]. Turski uses an analogy that nobody complains that the tyre goes flat, just that it happens at an inopportune moment. Accepting that flat tyres will happen, he argues the best policy is to make it easy to recover or cope with the failure.

Frederick Brooks is clear that the main difficulties in building software have nothing to do with the programming languages, development tools or other miscellaneous implementation details [Bro87]:

> "I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation."

This early stage is concerned with developing an appropriate specification for our problem. The issues are of an empirical nature and concerned with Brooks' "essential complexity". As Brooks explains in his 1987 article "No Silver Bullet" [Bro87], this is the fundamental nature of the problem to be solved:

> "The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. This essence is abstract, in that the conceptual construct is the same under many different representations. It is nonetheless highly precise and richly detailed."

Complexity in this form cannot be avoided, nor can it be abstracted away as it is an irreducible complexity. Beyond the essential complexity, computer programmers have struggled with issues that were not part of the original problem, issues that seemed to be introduced as part of the process of writing software. The decisions made at this stage are not intuitive or a priori reasoning but are based on solid experimental evidence and testing. Unless these decisions and experiments are based on familiar knowledge Vincenti terms this a "radical" design process [Vin93]. If we engage with a "normal" design mode, where the problem and environment may be familiar or regular, then it is possible for decisions to be made based on past experiences. With reference to the work of Naur [Nau85], the role of intuition in software development is considered in chapter 6. In this process of testing the conceptual idea of the software, it has not yet reached a deductive rational stage of reasoning.

The GC6 project is primarily concerned with the specification/program relationship. Specifically, it is focused on developing tools and techniques that allow a working implementation to be produced from a formal specification. Turksi and Maibaum label this a 'calculable' relation as it is possible to determine in finite time if the relation holds [TM87, p. 10]. The tools and techniques developed under the GC6 project aim to exploit the mathematically deductive nature of the relation to produce an implementation. In deductive reasoning, a sequence of statements are derived from the initial premise to reach a final goal. In this case, the initial statement is the specification and the final goal is the justification for a correct implementation. It is expected that, at this stage of the software development process, all conceptual decisions and experimentation has been completed. While some level of experimentation may still occur, this can only be in terms of

implementation technique.

While the GC6 project is concerned with program/specification relation, it is not primarily concerned with the problem/specification relation. As has been discussed, the problem/specification relation is not calculable and so is not amenable to formal deductive reasoning. The process of constructing a specification is primarily concerned with empirical observations and experimentation, and does not take place within a deductive rational framework. Where "radical" design occurs an environment is required to support exploration of conceptual ideas and empirical observations. This is difficult to achieve in the later stages of software development, where the deductive, rational part of the development process is emphasised. Formal methods, as used by GC6, are limited in this regard to an inherently rational view of software constrained to the calculable relationships.

1.6 Summary

This chapter examined the aims of the Grand Challenge on dependable systems evolution. It began by briefly reviewing the origins of the software crisis before moving on to an examination of the proposed verifying compiler. The potential benefits and implications of the use of these verification tools for software development were then considered. Finally, it questioned to what extent the challenge endorsed a rational view of software development.

Chapter 2

Formal Methods: Specification and Analysis

> "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."
>
> – C. A. R. Hoare

> "It's hard enough to find an error in your code when you're looking for it; it's even harder when you've assumed your code is error-free."
> – Steve McConnell [from Code Complete]

This chapter provides an introduction to the formal methods used to develop software. A brief historical background to formal methods is given, followed by an overview of some common formalisms that will be referenced in this thesis. Tools available to support these formal notations are also identified and discussed. Furthermore, this chapter will discuss how these notations and tools can be used in the process of software specification. The final sections of the chapter discuss

some of the common criticisms of formal methods.

The purpose of this chapter is to identify some of the common formal tools and techniques to allow for further discussion throughout this thesis. The chapter is not intended to be an exhaustive review of formal approaches to software development. Instead, the chapter examines the changing approaches to software development with formal methods and uses some examples to illustrate the discussion.

This chapter does not provide a detailed historical account of the development of formal methods. Background information on the circumstances of the software crisis and the history of the software industry can be found in [CK03] and [CKA04].

2.1 The changing role of Formal Methods

As chapter 1 explored, the software crisis of the 1960's manifested itself through software that did not meet its requirements, was not fit for purpose, and was not completed on schedule or within budget. Solving the problems identified during the software crisis remained a prominent goal for many years, occupying the efforts of many software researchers. Proposed solutions to the crisis included the development of a wide variety of software tools, development processes and methodologies. Yet none of these developments were completely successful. By 1987, Fred Brooks' "No Silver Bullet" [Bro87] article argued strongly that no individual technology would ever make a ten-fold improvement in productivity within 10 years. There would be no single technology capable of providing an all-encompassing solution to the challenges presented by software development. Software was a difficult problem and destined to remain so. Since the software crisis, the academic and industrial communities seem to have accepted the inherent difficulties in developing software – unable to rely on any one tool or technique. This thesis is primarily concerned with the first two manifestations – not meeting requirements and not being fit for purpose – of the software crisis and not with