Requirements, Traceability and Formal Software Development or a Further Analysis of Requirements Traceability*

Justin K. Pearson
Computer Science Department
Royal Holloway
University of London
Egham
Surrey TW20 0EX
e_mail: justin@dcs.rhbnc.ac.uk
Tel. +44 (0) 1784 443426
Fax. +44 (0) 1784 443420

October 1, 1996

Abstract

This paper is concerned with the interaction between formal software development and the issue of requirements traceability. The paper offers an analysis of the requirements traceability problem (see [1]) which takes into account software development using formal methods. Starting from the observation that formal software development is not an infallible method of producing error free and 'correct' code (see [2]); sources of error identified as in [3] are used to motivate a new analysis of the requirements traceability problem. Traceability is divided into: Pre-Requirements Specification, Pre-Formal Requirements Specification and Post-Formal Requirements traceability, a finer distinction than is made by Gotel and Finkelstein in [1]. This distinction is motivated by the problems associated with the process of formalizing a prose requirements specification document together with an

*The work described here forms part of the DATUM project (grant GR/H8994, project number IED4/19314) in the UK Safety Critical Systems Research programme supported by the EPSRC and DTI.

analysis of potential sources of errors in formal software development identified in [3].

1 Introduction

The formal methods community has for a long time recognized that it is naive to assume that given a formal specification, a fully working and *correct* program will appear solely by the use of formal methods. Perhaps the earliest example is the paper [2] concerning the way incorrect programs can be derived from incorrect or incomplete specifications. The observations in [2] arose from an exercise given to students, where the authors specified that a sorting program should be produced from a formal specification, but the specification omitted to include that the final sorted list must be a permutation of the original only that it is an ordered list. Consequentially any list would have been be a valid output of the program. While specification errors like this are easy to trap in small examples managing such 'obvious' conditions which are all to easy to omit in formal specifications can be hard on a large scale.

Requirements traceability (RT) is the ability to trace user requirements through the development process, from the requirements elicitation stage through to the final software product. The work of Finkelstein and Gotel in [4, 5] gives analysis of requirements traceability starting with empirical research through interviews, discussion and focus groups with practitioners in industry. It was found although people felt the need for requirement traceability they felt that there was a problem with what exactly constituted requirements traceability. The diagnosis offered by Finkelstein and Gotel was that there are two phases of requirements traceability, Pre-Requirements specification (Pre-RS) and Post-requirements specification (Post-RS)traceability (to be defined below) and that the essential problem lay in the Pre-requirements specification phase.

This paper builds on the analysis of [4, 5] and offers a further refinement of RT, when formal methods are used in development, into Pre-RS, Pre-Formal requirements specification and Post-Formal requirements specification.

While this work does not have the empirical backing of [4, 5] it offers a theoretical contribution based on the analysis of fallibility of formal methods as in [3] of what sort of data and problems constitute RT for formal software development.

This paper is structured into five further sections: section 2 is a short background to formal methods, section 3 and 4 gives an introduction to Gotel and Finkelstein's work on requirements traceability, section 5 is the main body of the paper which refines the classification of Requirements traceability to take into account formal methods, last there is a conclusion.

2 Formal Methods

As with Barroca and McDermid [6] the term 'formal methods' is used here to refer to methods which have a sound basis in mathematics. Which in terms of software development means that programs, environments and specifications are, or in principle, treated as mathematical entities. The treatment as mathematical entities allows transformations to be carried out which preserve the behavior and meaning of programs. Programs can be checked against specifications and can be shown to meet or fail to meet specifications. Examples of such methods include B [7] or VDM [8]. A distinction can be made between formal methods and rigorous methods. With rigorous methods, while still having a sound basis in mathematics, the degree of rigour in the proof steps is limited to the degree of rigour normally used in standard mathematical practise, while the degree of rigour in formal methods is much higher than would be normally expected in mathematics. An example, highlighting the difference between rigorous and formal mathematics, is the formalization of Landau's [9] analysis text book in the AUTOMATH[1] system of N.G. de Bruijn [10] (see [11]). Landau's text book was already of a much higher degree of rigour than would normally be expected in standard mathematical texts, but when the book was formalized in AUTOMATH, its length was increased ten fold. A historical example, the Principia Mathematica of Russell and Whitehead [12], which attempted to formalize the whole of mathematics, in terms of first order logic, this work took over 500 pages to assert that $1 + 1 = 2$. Also because of the length and complexity of the proofs, Principia Mathematica is full of mistakes. Worse still many of the theorems purported to be proved are actually false.[2] Judging from the Principia experience fully formal techniques are only practicable when there is adequate machine assistance.

Thus the use of rigorous methods, such as advocated by the VDM community (see [8]), is appealing, or is said to be appealing, because of lower development costs but mistakes can be made, e.g. steps in proofs such as "it is obvious that P" can be wrong and often are. For a detailed discussion of a rigorous proof of a Byzantine agreement protocol, that was formalized

[1]AUTOMATH is a system for implementing formal systems based on type theory.

[2]This was related to me by a college who knew a set theorist who was looking for exercises to set his class. When the set theorist looked through the later volumes of the Principia (which deals with higher cardinals) he found to his surprise many of the theorems false.

in the machine assisted formal system PVS and the subsequent errors found in formalization see [13]. The comments in this paper apply equally well to rigorous and formal methods. Because machine assistance is not feasible for rigorous proofs the possibility of error is greater. Therefore in rigorous development the use of RT is potentially more useful in uncovering errors, but more care would be needed in deciding exactly what constituted RT data because of the possibility of easly missed logical errors in development.

This paper assumes some familiarity with how formal methods might be applied, but no detailed knowledge is required of any particular formal method or technique (for an introduction see [3] or [14]).

3 Requirements Engineering

This section is intended only as a short introduction to Requirements Engineering (for a more detailed introduction see [15, 16]). A significant proportion of software development is usually done for other parties. To be able to write such software the developer must find out what the customer wants the program to do. The process of finding out what the customer wants is called Requirements Engineering. There are two important problems, which manifest themselves early on: the customer might not know what he or she wants, or the developer understands poorly what the customer wants due to poor domain knowledge.

The initial phase of requirements engineering is to elicit from the customer a set of requirements that the finished program has to satisfy and to produce a requirements specification (RS) document. This process involves feedback and many versions of the RS document are generally produced until an agreed final version is produced.

In a large number of cases the RS-document will be in natural language prose, possibly interpolated with pseudo-code to explain the operations and requirements of difficult sections[3]. In some cases it is desirable, or perhaps it is forced by contractual reasons, to have a RS-document expressed in a formal notation (such as Z [19], or VDM [8]). Such a document from now on will be referred so as a Formal Requirements Specification document (a FRS-document). The process of obtaining a formal requirements document

[3]In some requirements capture methods such as CORE [17] there is no provision for the recording of complicated functional requirements between data items. Often these functional requirements are specified in high level languages, which can force data representation at an early stage in the requirements, which is on the whole undesirable. The use of Z as in the SAZ method, which integrates formal methods in to a large software development context, [18] would improve matters.

from a prose requirements document is here referred to as the process of Formal Requirements Engineering.

It is to be emphasized that, at least in the author's opinion, the process of producing formal specifications should be in two stages. First a prose requirement specification document should be produced and then this document should be formalized. This formalization will produce a separate set of inconsistencies and queries to be resolved with the customer. See [20, 21] for examples.

For documented examples of producing FRS-documents (in this case Z specifications) see the book [22]. One of the largest examples in the book, is the formalization of parts of IBM's CICS transaction processing system. CICS is a system that has been under continuous development since 1969. In the early 80's it was decided that there would be some gain for maintenance and further development if the specifications of certain modules were formalized. Because there were existing working pieces of software and user manuals specifying the required behavior, this was a prime example of the process of Formal Requirements Engineering on a large scale.

The formal specification was obtained initially, by a process of feedback between experts in the Z notation and experts in the CICS system. Because the process of formalization forces the developer to think more clearly about the specification, inconsistencies were found in the original RS-specification (in this case the prose RS-specification was the user manuals and as an arbitrator there was the actual running code), which were either errors in the manual or bugs in the code. IBM originally had no intention to apply formalized refinement on the specifications but it was felt that the gain that would be achieved from just formalizing the specifications would be enough to warrant the investment in formal methods. The formal specifications were also validated by experienced users of formal methods, who commented on the style and content of the specifications. The reviewers were then asked to point out any inconsistencies with the original manual that they found. Many of these inconsistencies were due to the inexperience of the people writing the specifications, but some were real inconsistencies in the system. This process of feedback led to further refinements and revisions of the specifications. For a more detailed discussion, see Section IV of [22].

4 Requirements Traceability

The following definition is presented in [1]:

> **Requirements traceability** refers to the ability to describe and follow the life of a requirement, in both the forward and back-

wards direction (i.e. from its origins, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases).

After much empirical data gathering and analysis Gotel and Finkelstein [1, 5] refined RT into Pre-Requirements traceability and Post-Requirements traceability, again from [1]:

> **Pre-requirements specification (pre-RS) traceability**, which is concerned with those aspects of a requirement's life prior to its inclusion in the RS (requirement production)
>
> **Post-requirements specification (post-RS) traceability**, which is concerned with those aspects of a requirement's life that result from its inclusion in the RS (requirement deployment).

This refinement of RT was made in response to a general feeling that there was a RT problem, Gotel and Finkelstein identified the problem to be in the Pre-RS stage. They felt that Post-RS traceability issues were relativity tractable and were being addressed already.

This paper acknowledges that there still is a problem with Pre-RS traceability in traditional software development. While in formal software engineering, not only is there a Pre-RS traceability problem, there are a different set of requirement traceability issues in the Post-RS phrase due to the nature of formal methods. This will be elucidated in section 5.

Central to understanding the issue of traceability in formal software development is a refining of the Pre-RS and Post-RS traceability definitions from [1], to take into account the production of a formal requirements specification document. Thus, there will be three stages of traceability, Pre-RS, Pre-Formal-RS and Post-Formal-RS, The last two can be trivially defined, but are included for emphasis:

> **Pre-Formal-Requirements-Specification (Pre-FRS) traceability**, which is concerned with those aspects of a requirements life, prior to its formalization in a Formal Requirements Specification, and in particular how it is formalized. This is a separate stage from Pre-RS traceability
>
> **Post-Formal Requirements-Specification (Post-FRS) traceability**, which is concerned with the aspects of a requirements life that result from its inclusion in the Formal-requirements-specification.

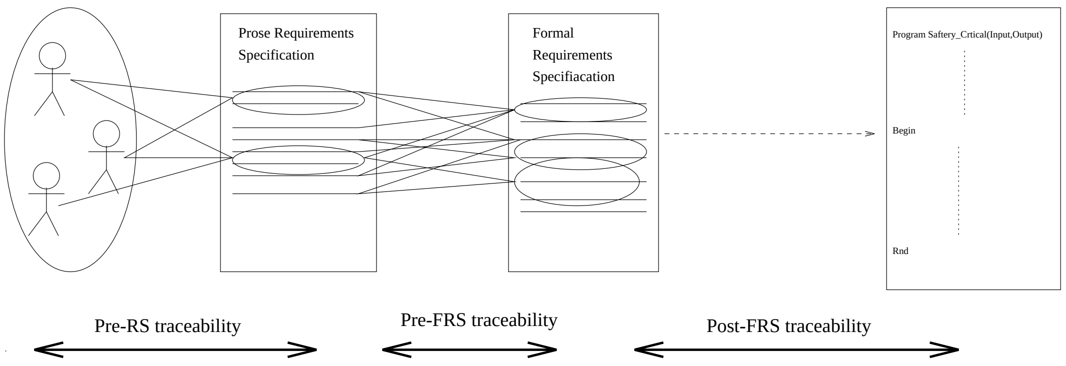
Prose Requirements Specification
Formal Requirements Specifiacation
Program Saftery_Crtical(Input,Output)
Begin
Rnd
Pre-RS traceability
Pre-FRS traceability
Post-FRS traceability

Figure 1: Pre and Post formal requirements specification

Figure 1 shows a pictorial representation of the situation.

As before the Pre-RS stage is gathering from the the customer to produce a RS-document. In the Pre-FRS stage the prose document is formalized and turned into a FRS-document. Again this stage should involve consultation with the customer to refine, understand the user requirements and to uncover any possible inconsistencies. Finally in the Post-FRS stage the FRS-document is used to refine or guide development of code. The analysis offered here, in the terminology of [1] is information driven: traceability is defined in terms of what information is to be made traceable, with special reference to the differences between Formal Software Engineering and traditional Software Engineering.

5 Use and Abuse of Formal Methods and Requirements Traceability

There are many sources of potential error in formal software development. For example in a fully formal development an obvious source is that if proofs are done by hand, mistakes can be made, typically because such proofs are normally long and tedious. Many other sources of error can be identified. Recent papers that address the issue of possible errors in software development include [23, 24, 25, 3]. I will concentrate on the sources of uncertainty identified in [3], and try to show how attention to such sources leads to a finer analysis of RT.

[3] identifies several classes of uncertainty in the use of formal methods: human ability, physical model, consistency of theories, tool quality and availability, process maturity, management pressure, and notation quality. The

following subsections summarize and expand Section 5 of [3] with a view as to how various sources of error contribute to the analysis of requirements traceability.

The issue of management pressure to produce deliverable software in shorter time scales, hence forcing corners to be cut while using formal methods (which can require longer time scales than in traditional software development), will not be discussed here. It is enough to say that a requirement for the successful use of formal methods is that a conducive management culture exists (and of course a conducive development culture). Of course, there is an 'extreme' view that software failures are really management failures [26]. Indeed one of the reasons cited for the success of the CICS example in [22] is that there was management backing for the use of formal methods. There are also wider management issues in the implementation of RT. This paper does not attempt to address these management issues in any depth. It simply offers a starting point with an analysis of RT for formal software development. One management issue that would have to be addressed include the management of RT data over distributed environments. But many (if not most) of the management issues are not primarily problems with formal software development and would have to be addressed in any project employing RT.

[3] talks about the problem of software process maturity with respect to the use of formal methods. Most individual formal methods at present do not cover the complete spectrum of the software development process. Integration with other techniques is often haphazard, (for an exception see [18]). If effective management of the software development process is not exercised with attention to the needs of formal methods then the quality of the final software can not be guaranteed. This paper is not going to address the wider management issues, but the issue of traceability of requirements is an attempt to enable the developer to understand how the quality and maturity of a particular formal method and associated tools affects the whole development process.

5.1 Human Ability

Software Engineering involves, at least to some degree, human beings, who are for the most part fallible. [3] rightly observes that the use of formal methods typically, at present, involves more human effort than traditional software engineering. Typically that is because formal methods force such a detailed level of analysis, so that many assumptions which would go unchallenged in traditional development have to be addressed. A further problem is that many software engineers are untrained in the sort of discrete mathematics needed for the effective use of formal methods. Therefore, the understanding of human ability is important in assessing the role of traceability in both formal and non-formal software development.[4]

[3] identifies two sets of uncertainties related to human ability, *knowledge* and *communication*. Knowledge includes three aspects:

(i) Lack of knowledge in the problem domain: the observation that the development process is more efficient when the programmer or system analysts understands the application domain.

(ii) Lack of knowledge in the development domain: the developer might not have solid knowledge of system development principles, nor have solid knowledge of formal techniques. Errors can result, there is no guarantee that formal methods will be applied correctly or effectively.

The lack of attention even to basic software engineering practice can have disastrous consequences, for example in the case of Therac-25, a safety critical application for administering radiation treatment to cancer victims, where there was no use of any software engineering principles, with a resulting loss of lives (see [29]).

(iii) Management might not have the knowledge to control the process of formal software development. This is part of a wider problem, that the process of producing software using formal methods is still not as well understood as traditional software engineering processes.

[3] divides communication, again into three aspects:

(i) Communication between the user and the developer may not be accurate, leading them to misunderstand each other during the acquisition of requirements. A problem directly related to formal software development, is that the user might not understand the formal specification produced.

(ii) Communication between individual developers on the project; in many situations there is more than one way to refine a specification, and developers may misunderstand the requirements and proceed on different refinement paths. As an example, suppose that the specification does not constrain the representation of a data type used, but does require that the final piece of code to be completely deterministic. A developer

[4]More recent studies [27, 28] have been carried out on the psychological classification of design errors, which influence the issue of traceability not only for formal software development, but also for traditional software engineering.

misunderstanding the user requirement for deterministic code might refine the data structures to a representation requiring dynamic memory allocation. The resulting code would be in general non-deterministic. This could be a perfectly formally correct refinement, but inappropriate for the user requirements (see section 3 of [14] for a discussion of refinement in formal software development methods).

(iii) Lack of Knowledge in the management domain. For example communication between developers and management, can result in needless errors being introduced.

Not all of the problems above are restricted to formal software development. Most importantly the communication aspects between the developer and the user in the requirements acquisition stage, is one of the aspects treated in [1] in their analysis of the Pre-RS traceability problem.

Both knowledge and communication based uncertainties affect Pre-FRS and Post-FRS traceability. First, with knowledge based uncertainties, lack of knowledge in the problem domain will affect Pre-FRS traceability requirements. The ideal (but unattainable) situation is that when the formal requirements specification document has been obtained development can take place without further interaction with the customer, of course in a world of changing requirements, this is not possible. Lack of knowledge in the problem domain on the designers part can result in inappropriate aspects being included in the Formal-RS document. To take a simple example, consider a system which is essentially a non-time critical batch processing system, formalizing real time constraints would detract from the intended required behavior of the system and complicate the specification unnecessarily. Lack of knowledge on the customer's part, where the customer does not understand the Formal-RS document fully, would again result in inappropriate aspects being included in a formal-RS document because the customer would not understand all the nuances of the specification produced. The use of independent consultants would be useful here, so as to review the formal specification produced. A Pre-FRS traceability regime would enable aspects of the Formal-RS document that were later found out to be inadequate or simply wrong, to be traced back to the informal user requirements and hence a better understanding of the software produced would be possible, which would make corrections and maintenance easier.

Knowledge based uncertainties affect Post-FRS traceability, mainly in the domain of lack of knowledge of the techniques involved. Inappropriate refinement can result in a useless program for the task at hand. If individual refinements are recorded and the reason why such refinements are done, audits can better assess the quality and reliability of the resulting software.

Communication based uncertainties will manifest themselves in any software project which has more than one person involved. The refinement example above is a communication based uncertainty that can be reduced by using Post-FRS. Recording the reasons why and how individual refinements reflect customer requirements is an important Post-FRS traceability requirement which can help uncover communication errors between developers.

Misunderstanding between the customer and the developer, due to inadequate communication in the Formal-RS document production stage, is an important problem to be solved. The understanding of how requirements evolve from informal to formal requirements can be improved by tracing the evolution of informal to formal requirements. Indeed the whole process of formalization is still not well understood. More research is required and the use of traceability data will be important input into the question of how requirements get formalized.

In summary lack of knowledge results in inappropriate aspects being included in the FRS-document. Pre-FRS traceability allows formal requirements to be audited and traced back to informal user requirements. Communication based uncertainties manifest themselves both before and after the production of the FRS-document. Problems of communication between the developers and the customers create similar traceability requirements as in knowledge based uncertainties. Problems of communication between developers on a project, create problems which could occur in any type development. But in formal development, if each developer has a different understanding of the Abstract system model (see below) then more adequate communication of concepts would be needed than in traditional development. The use of both Pre and Post FRS traceability allow communication based errors to be traced and understood.

5.2 Abstract System Model

In any software system, some model of the environment must be made, be it a banking system, an aircraft control system or a multi-tasking operating system. Various models can be made, and various approximations to reality used (see commandment II of [25]). The use of formal methods does not guarantee that the model of the software environment is any better than without the use of formal methods. But by expressing the attributes of environments formally, the designer is forced to think about things that might otherwise be missed and in many cases will produce a better model. Or sometimes produces a model of the environment where certain other software development methodologies fail to produce a model of the environment which goes outside expected interactions with the software. But the process of formal modeling has problems as well, sometimes the formal modeling process can impinge on reality and enforce unacceptable abstractions, see the discussion of [30] below.

Outside the problems with human error in modeling the environment, which has been discussed above under Pre-FRS traceability, further aspects of Pre-FRS traceability have to be taken into account. Modeling the software environment in a certain way is a design decision, e.g. the modeling of timing constraints or ignoring timing constraints affects the design considerations of the finished piece of code. A more concrete example would be the formalization of a requirement for continuous delivery of service. Depending on the method and style used some formalizations would already force certain design decisions. In the paper 'Symbol Security Condition Considered Harmful' [30] an analysis of the role of formal methods in security systems is given. Examples of formally verified systems which have gone wrong are presented (see below). But further, the paper offers an analysis of specification problems for secure systems. To understand the points made in [30], the process of modeling can be visualized as in figure 2. Schaefer produces many examples which can be seen as breakdowns of the top arrow in figure 2 (modeling assumptions). Schaefer believes that, if insufficient attention is played to the semantic nature of formal manipulations and the verification game becomes simply a symbol pushing game then the verifications produced can be next to useless. For example the use of fictional specification conveniences such lumping together hardware registers or adding adding extra reference conditions to simplify proofs of systems. While sometimes these maybe valid moves, their blind and unchecked use can lead to the introduction of security flaws in formally verified systems. See [30] again for formally verified systems which were later found to contain security flaws[5]. Another problem, alluded to in [30], in the security community is the use of off the self models. The use of formal methods was mandated very early (1970's) on for secure systems. At the time there was not much experience of formal specification and verification. Bell and La Padula introduced a model [31] of security access in file-type systems. While this model covered a large number of cases, its popularity caused some designers to simply take the model and attempt to fit it to the situation at hand. Consequently it was applied to inappropriate systems. For example while it is generic for file systems when applied to database systems it already forces some design considerations which are unnecessary. The use of Pre-FRS would allow mistakes like this to be found,

[5]Perhaps it should be stressed that Schaefer message for formally verified systems is not all negative, he believes that their use is beneficial, but more understanding of their limitations, quirks and pitfalls are required.

and further if applied properly it would force specifiers to think about why things are done in certain ways.

User requirements need to be traced through to the formal-requirements specification document, with a view to understanding how the user requirements are reflected by the modeling technique, and differentiating them from modeling decisions that are just a matter of style.

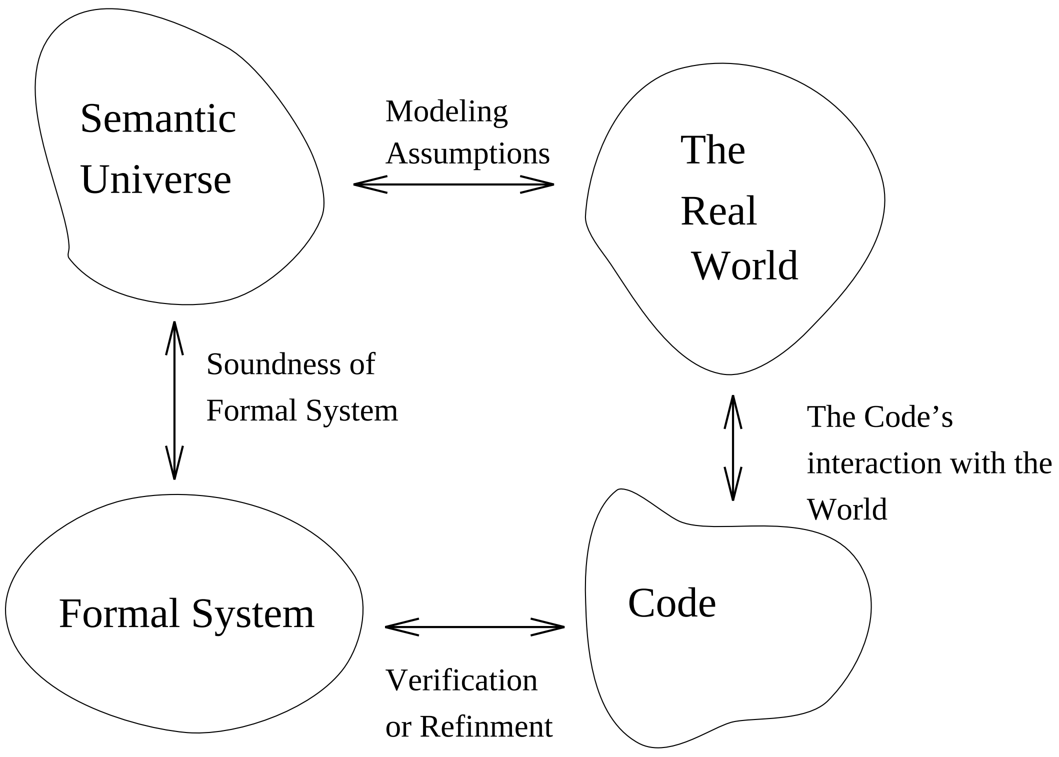
Semantic Universe
Modeling Assumptions
The Real World
Soundness of Formal System
The Code's interaction with the World
Formal System
Code
Verification or Refinment

Figure 2: The Process of Abstraction

5.3 Consistency of Theories (and Methods)

If the formal theory used in implementing formal software development is not consistent, then the quality of the final piece of code is not guaranteed[6]. It is surprising how easy it is to construct proofs based on inconsistent assumptions which appear not to be consistent. In [30] a case is quoted of a data access control system AUTODIN II [32] which was supposedly formally

[6]For any formal system a contradiction can be used to prove anything, thus if $0 = 1$ is introduced as an axiom then every possible statement is true in the formal system.

verified, later the specification was found to be inconsistent. Essentially a pre-condition to a state transition stated that a component of the state invariant should have two contradictory values.

More worrying, it is surprising how many implemented proof systems are, or were, inconsistent. This means it is not sufficient a piece of software has been proved correct by mathematical methods with or without tools. What needs to be said is how it was proved correct. This need not be a list of every proof step, but at least enough to reconstruct the relevant proofs, possibly relative to the software tool support used. With the AUTODIN II example above, many of the proofs would go through, but those relying on the contradictory state transition would be false. Traceability in the Pre-FRS stage would allow reasons to be uncovered as to why the contradictory state transition was introduced and Post-FRS traceability would enable one to uncover the effect of the contradictory state on the correctness of the verification. Further, proofs are done for a reason and how the proofs affects the user requirements needs to be recovered, e.g this process has been guaranteed to terminate in less than 10 seconds of cpu time, because the user requirements specified that a result had to be produced in a 'reasonable' amount of time.

5.4 Tool Quality and Availability

Because in a formal verification every theorem must be proved and every step justified, there are many proof obligations that have to be satisfied. Doing this by hand is a tedious and error prone task. One of the major criteria that many industrial users of formal methods require is that the formal method in question has adequate automated tool support.

Tools for formal methods support will themselves typically be large pieces of software, and will in general contain software errors. Thus, one cannot completely rely on the output of an automated theorem prover as an infallible certificate of reliability. Also some tools such as the B-Toolkit[7] offer automated proof environments with a user driven proof tool for when the automated proof tool is unable to prove the goals. Unfortunately, at least in the early versions, the user directed proofs were so non-standard it was easy to introduce logical inconsistencies in the proofs[8]. Such proofs produced by the tool would have to be audited and the use of RT would help the audit trail. But tools are being improved and certain tools will be perceived to be more reliable than others. Aspects of the post-FRS traceability regime must include how the rôle of tool support implements user requirements. For

[7]Trade mark B-core U.K.

[8]Thankfully later versions are improved.

example, a particular tool might be used to prove that the CSP specification of the system satisfies certain security properties, because there is the requirement X in the RS-Document.

Furthermore, since for a particular formal development procedure there might not exists tools to cover the entire spectrum of software development activities, the choice of method (Z, VDM CSP, etc.) will be affected, depending on the user requirements. This is a Pre-FRS requirements traceability problem, i.e. how the user requirements are reflected in the choice of formal methods used relative to the available tool support.

5.5 Notation Quality

Different notations have different strengths. Modeling real-time behavior with a model based approach such as Z or VDM, would be possible, but difficult (see commandment I of [25] for a fuller discussion). Further, changing the representation of a problem can make it easier to understand and solve. As with the traceability requirements for the choice of abstract system model, a Pre-FRS traceability issue has to be addressed, i.e. how does the choice of notation reflects the user requirements?

6 Summary and Conclusion

This paper has investigated various possibilities for error in formal software development, and has begun to show how the rôle of traceability is not to remove errors, but to allow these errors to be found and corrected more quickly when discovered and hence to lead to a better quality environment for the production of high reliability software. Further traceability in general allows the developer to cope with and understand the demands of changing user requirements during the project. However there is little empirical evidence that the use of quality oriented techniques in software development will lead to more reliable software. Gotel and Finkelstein [4] discuss why quality methods have not led to the dramatic improvement in software quality that might have been expected. In their opinion it is because of a lack of understanding of the Pre-RS stage of development and they believe that Pre-RS traceability is fundamental to improving the quality of software. As a corollary to this, if the analysis offered here of traceability in formal software development is in anyway correct, there should be an improvement in the quality formally developed software.

One of the criticisms often given with respect to quality procedures is that they generate too much paper work. Making the quality process as painless as possible and as paperless as possible is an important goal to be pursued for the success of any quality development procedures. Gotel and Finkelstein [1] divide the available tool support into four areas: General purpose traceability tools, such as hypertext editors, word processors or databases; special purpose tools, such as tools designed for requirements engineering such as RTM of Marconi, which provide some support for requirements traceability; workbenches i.e. integrated tool support to cover many aspects of the development cycle; and software environments, which are integrated environments typically centred around a database model designed to cover the whole of the development process. To the author's knowledge none of the special purpose requirements tools available is specifically addressed to the use of formal methods. Further, because the use of formal methods does not cover the whole of the software development life cycle, it seems that integration of available tools in workbenches and environments is desirable. Gotel and Finkelstein [4] discuss the relation between modeling the artifacts used in the requirements engineering process (such as programs, pieces of paper, faxes etc.) and databases of personnel and groups involved in the design process. Gotel and Finkelstein examine the semantic nature of the relations between requirements, artifacts and people, and propose that a more detailed semantic relation is needed than the statement "X contributed_to Y". Further research is needed on how the nature of the uncertainties in formal software development identified above affect the semantic data that needs to be recorded. The use of languages such as SGML, which annotate the semantic nature of documents, has already been applied to Z (see Annex D of [33]). This could be expanded to produce a standard interchange format for traceability requirements data and in itself requires further research, as well as how such data can be integrated in a relational database. This whole framework would to be codified in an open structure so that tool builders can easily hook into a standard traceability environment.

This paper has given a further analysis of RT, refining the work of Gotel and Finkelstein [4, 5]. The analysis has been driven by potential sources of error in formal software development as identified in [3]. This analysis is merely a start it is intended to be both a contribution to the formal software development community and to requirements traceability. The use of traceability in formal software development will hopefully contribute to the validation and management of large projects. On the traceability side it has been shown has the special nature of a certain software process can contribute to the analysis of RT. A similar programme could be carried out for other software methodologies, such as object oriented technologies. Such a programme would contribute to the general understanding of how the software process effects requirements traceability.

7 Acknowledgments

The author wishes to thank many people who have read early versions of the paper. Including Norman Fenton, Steve Schneider, Dieter Gollmann and Gordon Rugg.

References

[1] Orlena C.Z. Gotel and Anthony C.W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the IEEE International Conference on Requirements Engineering (ICRE '94)*, pages 94–101, Colorado Springs, Colorado, April 1994.

[2] S. Gerhart and Yelowitz L. Observations of fallibility in applications of modern programming methodologies. *IEEE Transactions in Software Engineering*, SE-2(3):195–207, 1976.

[3] Shaoying Liu, Victoria Stavridou, and Bruno Dutertre. Formal methods and dependability assessment. In *Proceedings of Compass 94*, Wasington DC, June 1994.

[4] Orlena C.Z. Gotel and Anthony C.W. Finkelstein. Modelling the contribution structure underlying requirements. In *Proceedings of the First International Workshop on Requirements Engineering: Foudation of Software Quality (REFSQ '94)*, pages 71–81, Utrecht, The Netherlands, June 1994.

[5] Orlena Cara Zena Gotel. *Contribution Structures for Requirements Traceability.* PhD thesis, University of London, Imperial College, 1995.

[6] L.M. Barroca and J.A. McDermid. Formal methods: Use and relevance for the development of saftey-critical systems. *The Computer Journal*, 35(6):579–599, 1992.

[7] Kevin Lano. *The B Language and Method: A Guide to Practical Formal Development.* FACIT. Springer Verlag, May 1996.

[8] C. Jones. *Systematic Software Development using VDM.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1986.

[9] Edmund Landau. *Grundlagen der Analysis.* Akademische Verlagsgesellschaft, Leipzig, Germany, 1930. English translation *Foundations of Analysis*, Chelsea Publishing Company, 1951.

[10] N.G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In M. Laudet, editor, *Proceedings of the Symposium on Automatic Demonstration*, pages 29–61, Versailles, France, December 1968. Springer-Verlag LNM 125.

[11] L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH System.* PhD thesis, Eindhoven University of Technology, 1977.

[12] B. Russell and A.N. Whitehead. *Principia mathematica.* Cambridge University Press, 1910-13.

[13] J. Rushby and P. Lincoln. A formally verified algorithm for interactive consistency under a hybrid fault model. Technical Report CSL-93-02, SRI International Computer Science Laboratory, March 1993. Also available as NASA Contractor Report 4527, July 1993.

[14] Doanld Sannela. A survey of formal software development methods. In A. McGettrick and R. Thayer, editors, *Software Engineering: A European Prospective*, pages 281–297. IEEE Computer Society Press, 1993.

[15] Roger S Pressman. *Software engineering : a practitioner's approach.* New York : McGraw-Hill, 1987.

[16] Hubert F. Hofman. Requirements engineering. Technical report, Institut für Informatik der Universität Zürich, März 93.

[17] Anthony Finkelstein and Jeff Kramer. TARA: Tool assisted requireements analysis. In P Loucopoulos and R. Zicari, editors, *Conceptual Modelling, Databases and CASE: an intergrated view of information systems development*, pages 413–432. John Wiley, 1991.

[18] Fiona Polack and Keith C. Mander. Software quality assurance using the SAZ method. In J. Bowen and J.A. Hall, editors, *Z User Meeting Cambridge*, pages 231–249. Springer Verlag, 1994.

[19] J.M. Spivey. *The Z notation : a reference manual.* Prentice-Hall International Series in computer science. Prentice Hall, 1989.

[20] J.S. Fitzgerald, T.M. Brookes, M.A. Green, and P.G. Larsen. Formal and informal specifications of a secure system component: First results in a comparative study. In Maurice Naftalin, Tim Denvir, and Miquel Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecutre Notes in Comuter Science*, pages 35–45. Springer-Verleg, 1994.

[21] Joshua D. Guttman and Dale M. Johnson. Three applications of formal methods at MITRE. In Maurice Naftalin, Tim Denvir, and Miquel Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecutre Notes in Comuter Science*, pages 35–45. Springer-Verleg, 1994.

[22] I. Hayes. *Specification Case Studies.* Series in Computer Science. Prentice Hall, second edition, 1993.

[23] J.A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[24] J Bowen and M Hinchey. Seven more myths of formal methods. Technical Report 357, University of Cambridge Computer Labratory, 1995. To appear in IEEE Software.

[25] J.P. Bowen and M.G. Hinchey. Ten commandments of formal methods. Technical Report 350, University of Cambridge, 1994.

[26] A. Wingrove. Software failures are management failures. In B. Littlewood, editor, *Software Reliability, Achievments and Assessment*, pages 56–68. Blackwell Scientific Publications, 1989.

[27] Alistair Sutcliffe and Gordon Rugg. A taxonomy of error types for failure analysis and risk assessment. 1994.

[28] Alistair Sutcliffe, Gordon Rugg, and Peter Ayton. Pitfalls in the design process: Assessing the potential for experts' errors. Draft to be submmitted to J.High Integrity Systems (1994).

[29] N.G. Leveson and C.S. Turner. An investigation of the THERAC-25 accidents. *Computer*, 26(7):18–41, 1993.

[30] Marvin Schaefer. Symbol security condition considered harmful. In *1989 IEEE SYMPOSIUM ON SECURITY AND PRIVACY*, pages 20–46. IEEE, 1989.

[31] D.E. Bell and L.J. La Padula. Secure computer systems. Technical Report 2547, The MITRE Corporation, May-Dec 1973 1973. vol I-III.

[32] S. Bergman. A system description of AUTODIN ii. Technical report, MITRE Corporation, Bedford, Mass, May 1978.

[33] Z Standards Review Committe. Z base standard, version 1.0. Technical report, PRG Group Oxford, 1992.