

An Investigation of the Laws of Software Engineering

Paul Watkinson, Kellogg College, Oxford University, 31/7/2011

Paul.watkinson @ Kellogg.ox.ac.uk

Abstract

We have reached the point where we are often completely dependent on software systems. Software Engineering isn't based on fundamental laws like Chemical and Mechanical Engineering. Defining a Software Engineering Methodology which gets to the root of the problem, through going back to first principles and identifying the fundamentals laws by revisiting the software mathematical principles should resolve this issue. A new Software Engineering methodology is proposed of applying "Natural Deduction" and the "Axiom of Reducibility" to the elicitation and verification of requirements responsible for the logical control of software systems, and allowing automatic programming of a verifiable computation.

1 Introduction

1.1 Identification of problems

"The object of science, properly so called, is the knowledge of laws and relations. To be able to distinguish what is essential to this end, from what is only accidentally associated with it, is one of the most important conditions of scientific progress." George Boole, 1854, from "An Investigation of the Laws of Thought - On Which are Founded the Mathematical Theories of Logic and Probabilities". [12]

More than ever before, the failure of a software system may have severe consequences for our safety, health, our livelihood, or our society. [19] One of the many reasons for this issue is the inaccurate capturing and interpretation of requirements. Alan M. Davis stated regarding the Software Requirements Specification (SRS) that *"If a perfect SRS could exist it would be: correct; unambiguous; complete; verifiable; consistent; understandable by the customer; modifiable; traced; traceable; design independent; annotated; concise and organised".[7]* The "Natural Deduction" approach sits between Formal Methods like Z and informal diagramming techniques such as UML and ensures decidability. It addresses the issue of producing key requirements so that they have the same meaning to Business Stakeholders and the computer. The "Natural Deduction" methodology is applied to a security case study of "Hotel Locking" and compared with alternative approaches.

1.1.1. Software Abstractions

Daniel Jackson, Professor of Computer Science at MIT in his book “*Software Abstractions: Logic, Language, and Analysis*” defined the following Hotel Room Locking problem and then set about using a tool that he and his students had designed and developed called “Alloy” based on Z to model it. Interestingly he also asked the leading experts in Formal Methods to model it as a comparison in B, OCL, VDM and Z. This gives the basis for a comparison of methods.[1]

“Hotel Room Locking

Most hotels now issue disposable room keys; when you check out, you can take your key with you. How, then, can the hotel prevent you from re-entering your room after it has been assigned to someone else? The trick is recodable locks, which have been in use in hotels since the 1980’s, initially in mechanical form, but now almost always electronic.

The idea is that the hotel issues a new key to the next occupant, which recodes the lock, so that previous keys will no longer work. The lock is a simple, stand-alone unit (usually battery-powered), with a memory holding the current key combination. A hardware device, such as a feedback shift register, generates a sequence of pseudorandom numbers. The lock is opened either by the current key combination, or by its successor; if a key with the successor is inserted, the successor is made to be the current combination, so that the old combination will no longer be accepted.

This scheme requires no communication between the front desk and the door locks. By synchronizing the front desk and the door locks initially, and by using the same pseudorandom generator, the front desk can keep its records of the current combinations in step with the doors themselves.” [1]

1.2 Objectives of study

To provide an academic analysis of a concrete problem, Hotel locking. This problem has security, safety and economic factors.

1.3 Organisation of the study

Natural Deduction which is a generic logic and the “Axiom of Reducibility” will be used for requirements gathering as a security case study with the hotel lock problem. There will be a comparison with other approaches of requirements elicitation.

2. Literature Review

In large projects there are often a set of key business rules which define the scope of the project. The example used in this dissertation is a classic case. Even though the implementation of the full Hotel Locking system will require a number of components and take some time to implement, it can essentially be described in one page and translated into a

simple set of rules [1]. The main question is what approach to use to represent these rules? Ideally the rules should be both human and machine readable, and after some investigation and comparison of options “Natural Deduction” was identified. It has the benefit of being very simple and precise. It was also felt that with today’s modern user interfaces it should be possible to design and develop a software program which would help the users ensure that the business requirements were written correctly in “Natural Deduction” and follow the “Axiom of Reducibility”. (Please check www.km2wd.com for further developments)

The term “Natural Deduction” was coined independently by Gentzen and Jaśkowski in 1935. This was two years before Alan Turing wrote his classic paper “On Computable Numbers, with an application to the Entscheidungsproblem” which described the computer[6]. During the period of 1935 to 1939 Gentzen was an assistant of Hilbert who in 1900 had defined the “Entscheidungsproblem”, which was regarded as a concrete mathematical problem of undecidability. While Gentzen identified “Natural Deduction”, Turing’s paper is based on the principle of how someone would use “Natural Deduction” to compute real numbers. Turing broke down the steps that someone uses to add up some numbers for example and abstracted the basic rules and provided a proof. The resulting method was also shown to be able to prove any logical formula and hence the term “Turing machines” to mean a machine that can compute any formula, or the computer as we know it today. The following statement is from Turing’s paper *“In other words, we assume that there is an axiom U which expresses the rules governing the behaviour of the computer, in terms of the relation of the state formula at any stage to the state formula at the preceding stage. If this is so, we can construct a machine to write down the successive state formulae, and hence to compute the required number.”*[6]

The key factor is that the computer is using a very simple logic called “Natural Deduction” and any logical rule can ultimately be defined within it, barring the paradoxes. There are many forms of logics: C and java are both logics, the problem is that they mix data and control logic and thus hide the control part to the untrained eye. What “Natural Deduction” does is to isolate the control part. This is the component in a computer program that can make it so difficult to test all of the combinations.

Aristotle was the first to identify the primitive forms of logic for the purpose of law. Boole identified that true or false could be translated to 1 or 0 which put in place the concept of the digital system.[12] Peano converted Aristotle’s rules for numbers into a set of axioms which define for example the sequence “0123456”, i.e. that they increment by one and start at 0 etc. Russell Identified that Peano had missed a rule that was caused by a paradox that had been identified earlier by Lewis Carroll [8] which affects “IF-THEN” clauses and was addressed by Russell and Whitehead in “Principia Mathematica” with the “Axiom of Reducibility”.[9]

Richard Bornat showed how “Natural Deduction” was a common subset of different logics and had essential control attributes, that consists of just four connectives “Conjunction”, “Implication”, Disjunction” and “Negation” as shown in Table 3 [3]. Bornat stated in his “Proof and Disproof in Formal Logic” that *“This part of the book is about formal proof in a particular logical system, Gerhard Gentzen’s simple and beautiful **Natural Deduction**, developed in 1935. As a result of Gentzen’s brilliant work, there isn’t much to know about*

Natural Deduction – just a few symbols, each with two or three rules...the connectives, which are like the operators of arithmetic...the quantifiers, which are like procedures, functions or methods in programming languages.” [3]

Natural Deduction			
Connective Symbol	Ascii Form	Simple Name	Latinate Name
\wedge	&	And	Conjunction
\rightarrow	-->	Arrow, if-then	Implication, conditional
v	v	Or	Disjunction
\neg	~	Not	Negation

Table 3: The four forms of “Natural Deduction” [3]

James Martin showed how primitive logical connectives could be used to ensure provably correct constructs and automatically produce verifiable code. He stated *“To prevent software from ‘conking,’ it needs to be based on constructs which are mathematically provably correct, and modifiable with the same techniques, so that correctness is preserved. These techniques are too tedious for pencil-and-paper design. They require computerized tools. The tools must automatically generate bug-free programs because we cannot expect human beings to do so.”[5]*

Jim Woodcock and Jim Davies stated of Z’s logical language *“Collected together, these rules form a system of natural deduction: they state what may be deduced from a proposition, and under what conditions that proposition may be concluded. This provides a framework for reasoning about statements in our language, proving properties and establishing results.” [4]*

The Natural Deduction defined by Bornat (Gentzen) in Table 3 is a subset of that used by Z which excludes “equivalence” (if and only if). Martin’s approach uses the three primitives of “JOIN”, “INCLUDE” and “OR” although in the paper that he bases this on, written by Margaret Hamilton and Saydean Zeldin their proof uses the Natural Deduction connectives. Turing did not explicitly state “Natural Deduction” in his paper but used the connectives of Natural Deduction in his workings and examples. “Natural Deduction” is the commonality between all of these systems, which is why a conclusion was reached that this method was significant.

3 Methodology

The “Natural Deduction” methodology uses “only” the following connectives of propositional logic “Conjunction”, “Implication”, “Disjunction” and “Negation”. [7.2 Appendix 2 – Patent 2] In this dissertation there is a particular focus on separating the concern of “control” within a programming language. Why is the propositional calculus not extended to predicate calculus? Michael Huth and Mark Ryan state *“Predicate logic is much more expressive than propositional logic, having predicate and function symbols, as well as*

quantifiers. This expressiveness comes at the cost of making validity, satisfiability and provability undecidable.”[13p136] Huth and Ryan go onto state “*Coding up complex facts expressed in English sentences as logical formulas in predicate logic is important – e.g. in software design with UML or in formal specification of safety-critical systems – and much more care must be taken than in the case of propositional logic.*”[13p95] Turing was very much concerned with decidability in his 1937 paper “On Computable Numbers...” when he defined the “Turing machine”. However in his later Phd thesis in 1938 under Alonso Church “Systems of logic based on Ordinals” Turing states “*In consequence of the impossibility of finding a formal logic which wholly eliminates the necessity of using intuition, we naturally turn to “non-constructive” systems of logic with which not all the steps in a proof are mechanical, some being intuitive. An example of a non-constructive logic is afforded by any ordinal logic.*” Turing is saying that “Natural Deduction” can never be entirely formalised there has to be a human element of intuition - the action of writing the statement. Once the statement has been written as a Well Formed Function (W.F.F.) then it can be interpreted by the computer - a “Turing machine”.

Turing wrote a paper “The Use of Dots as Brackets” in 1942 which expanded on the “Vicious Circle” problem although the paper is incomplete.[6] Russell and Whitehead’s “Axiom of Reducibility” ensures that the brackets are correctly applied as a “hierarchy” to avoid the “Vicious Circle” [7.1 Appendix I – Patent 1]. It is believed that using Jan Łukasiewicz’s technique of “Reverse Polish Notation” which eliminates bracketing provides such a proof, where the statement is broken into a series of statements like “plates” and put one on top of the other, forcing a hierarchy.

In Turing’s paper “On Computable Numbers” he talks about the “restricted Hilbert functional calculus” the aim of which was to achieve decidability, which is interesting because from this we can derive that he had read Hilbert and Ackerman’s “Principles of Mathematical Logic” the first edition. This is because in the second edition the preface states that the term “functional calculus” has been replaced everywhere by “predicate calculus”. It also interestingly states that “*It was possible to shorten the fourth chapter inasmuch as it was no longer necessary to go into Whitehead and Russell’s ramified theory of types, since it seems to have been generally abandoned.*” This might explain why Turing left it out of his paper. I believe that the “Ramified Theory of Types” which relates to the “Axiom of Reducibility” is significant and should be included, hence patent 1 in the Appendix. [Appendix 1] It also states in the same preface that “I am also grateful for various suggestions to Mr. G. Gentzen of Gottingen .. Burgsteinfurt, November, 1937”. This date is significant because it was after Turing’s paper was published which means that Hilbert, Ackerman and Gentzen could well have read Turing’s paper before writing this. [21][18]

To explain the significance of this “Natural Deduction” methodology to Software Engineering, TOGAF is an Architecture Methodology developed by the Open Group [2][10]. In the TOGAF Framework it states “*The ability to deal with changes in requirements is crucial. Architecture is an activity that by its very nature deals with uncertainty and change – the “grey area” between what stakeholders aspire to and what can be specified and engineered as a solution. Architecture requirements are therefore invariably subject to*

change in practice. Moreover, architecture often deals with drivers and constraints, many of which by their very nature are beyond the control of the enterprise (changing market conditions, new legislation, etc) and which can produce changes in requirements in an unforeseen manner.” [2] The concept of using “Natural Deduction” for the elicitation of requirements could be particularly relevant to “Architectural requirements” described in the TOGAF methodology, as these are often describing logical control problems.

Richard Fairley in his book “Software Engineering Concepts” states “*The goal of software requirements definition is to completely and consistently specify the technical requirements for the software product in a concise and unambiguous manner, using formal notations as appropriate.*”.[16] He goes on to say in a chapter on “Verification and Validation Techniques” that “*Major theoretical limitations are imposed on static analysis by decidability theory. Decidability results can be phrased in various ways and have many profound implications. One phrasing states that, given an arbitrary program written in a general purpose programming language (one capable of simulating a Turing machine), it is impossible to examine the program in an algorithmic manner and determine if an arbitrary chosen statement in a program will be executed when the program operates on arbitrarily chosen input data. By ‘algorithmic manner’ we mean that it is impossible to write a computer program to perform this task for all possible programs. Furthermore there is no algorithmic to identify all programs and data for which the analysis is possible; otherwise the halting problem for Turing machines would be solvable.*

Decidability theory thus states that it is not possible to write a static analyser that will examine an arbitrary program and arbitrary input data and determine whether the program will execute a particular halt statement, or call a particular subroutine, or execute a particular I/O statement, or branch to a given label. It is therefore impossible to perform static analysis on an arbitrary program and automatically derive a set of input data that will drive the program along a particular control path. It is possible, however, to place restrictions on programs so that undecidable issues become decidable.” [16] The use of Natural Deduction effectively ensures that the requirement is decidable, an important step towards reliable software. Hamilton and Zeldin stated “It is inconceivable that mathematicians would solve or communicate mathematical problems without the formalization of mathematics or the language of mathematics. Yet software has been developed without its own consistent set of formal laws or a software meta-language of its own.”[22] Hamilton and Zeldin from their experiences of designing the software for the Apollo moon landings identified that it was very important to ensure the program was decidable and went to further lengths when producing the Shuttle control software of providing a framework which ensured that the designers could not write an undecidable statement. They achieved this by defining a primitive language and used a tool to ensure that it was correctly applied.[22] They were effectively applying Natural Deduction at the most basic level to the problem of writing correct software.

In Turing’s paper he refers to a “computable sequence” saying that “the terms of \mathcal{U} must include the necessary parts of the Peano axioms ...”. Turing refers to Godel’s paper “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I” as

“superficially similar” which has a footnote referring to the “Axiom of Reducibility”. Turing refers to the “Principia Mathematica” axioms elsewhere within his paper so he was familiar with Bertrand Russell and Alfred Whitehead’s book particularly as they were also from Cambridge University. The “Axiom of Reducibility” is used to address the vicious-circle principle and define a hierarchy of functions. It is possible that Turing was aware of this but didn’t consider it necessary to explicitly document it in his paper. Either way the “Axiom of Reducibility” still has not been implemented in the computers we are familiar with today.

Did John Von Neumann when he defined the computer architecture still commonly used deliberately or accidentally miss interpret Turing’s paper? We need to appreciate that when Turing wrote the paper he was only 24 and it was a work of genius. He had no computers or even electronics to base it on, it was purely theoretical, and he assumed that it was a process that a human would undertake as if they were a machine. He may also not fully have considered the very large programs that we have today with millions of instructions, because his idea turned out to be so useful and popular. Turing wrote the paper for his peers, people who understood the subject he was talking about so maybe he felt he did not need to explain everything. He states “It is assumed that the terms ‘Well-formed formula’ (W.F.F.) and ‘conversion’ as used by Church and Kleene are understood” which was added later in August 1936 after he joined The Graduate College, Princeton University, New Jersey in the USA to study for a Phd under Professor Alonso Church. John Von Neumann, Albert Einstein , Max Newman were there at the same time and discussed the field and John Von Neumann offered Turing a job as his researcher. Church had written papers early in 1936 “An Unsolvability Problem of Elementary Number Theory” and “A Note on the Entscheidungsproblem” that had been sent to Max Newman, Turing’s professor at Cambridge and which Turing based his paper on.

The essence of the “Axiom of Reducibility” is that it defines a hierarchy like a tree similar to the tree in the forest or a family tree that we are all familiar with. Russell gives an example of “Napoleon had all the qualities that make a great general” and it was these “qualities” that should form a hierarchy to be correct. The author has registered a patent [7.1 Appendix I – Patent 1] to apply the “Axiom of Reducibility” to the software within the hardware to ensure that only programs which satisfy the hierarchy can be run successfully, thus defining a modification to the John Von Neumann architecture. All programs would be pre processed and analysed to ensure they satisfy the “Axiom of Reducibility” before being run. This would be the equivalent of adding an additional step to Turing’s original paper of processing the tape and confirming that the program is a W.F.F. before running it through the “Turing Machine”. The result should be a new paradigm for computers so that they can be trusted to be W.F.F. and therefore scientific approaches can be applied to the design of software for them. The software developer does not need to rely just on intuition to ensure that the program looks ok, if it isn’t their program will not run at all, and we in our cars and planes will be safer as a result.

The principle rules for the computer outlined by Turing are not written down in one place and are incomplete; they may also have been restricted for security reasons at the time due to World War II. They are spread over a number of papers by Turing, Godel, Peano, Russell,

Post, Einstein, Bernays, Hilbert, Von Neumann etc. They all use different mathematical notation and sometimes language such as French, German, Italian and English. This has been a common problem, Edgar Codd who was educated at Exeter College, Oxford, and invented the Relational Database said of Vendor's not implementing his design fully "One of the reasons they offer is that they cannot collect all of the technical papers because they are dispersed in so many different journals and other publications". Tim Berners-Lee who invented the World Wide Web and studied at Queens College, Oxford, had issues with CERN and the copyright of the idea.

The following table contains the Axioms of Turing's 'Well-formed formula' (W.F.F.) which is the part of Turing's thesis which relates to the well formed-ness of a "Computer Program", from Russell's 1903 "The Principles of Mathematics" (pages 16&17).[18]

#	Axiom	Rule	Description
1		If p implies q, then p implies q	Whatever p and q may be, "p implies q" is a proposition
2		If p implies q, then p implies p	Whatever implies anything is a proposition
3		If p implies q, then q implies q	Whatever is implied by anything is a proposition
4		If p implies p, then, if q implies q, pq (the logical product of p and q) means that if p implies that q implies r, then r is true.	If p and q are propositions, their joint assertion is equivalent to saying that every proposition is true which is such that the first implies that the second implies it
5	Simplification	If p implies p and q implies q, then pq implies p	The joint assertion of two propositions implies the assertion of the first of the two.
6	Syllogism	If p implies q and q implies r, then p implies r.	
7	Importation	If q implies q and r implies r, and If p implies that q implies r, then pq implies r.	
8	Exportation	If p implies p and q implies q, then, if pq implies r, then p implies that q implies r.	
9	Composition	If p implies q and p implies r, then p implies qr	A proposition which implies each of two propositions implies them both.
10	Reduction	If p implies p and q implies q implies q, then " ' p implies q ' implies p" implies p	Axiom of Reducibility.

John Von Neumann based his architecture for the computer on a paper by McCulloch and Pitts which was written during the Second World War and analysed how the brain's neurons could be mapped to logic. This in turn took Turing's paper as its basis. It therefore seems

appropriate to revisit this area and look to see if something was missed. Natural Deduction and the Axiom of Reducibility may provide that answer.

4 Results, Discussion and Analysis

4.1. Hotel Room Lock Problem, definition.

4.1.1. Natural Deduction

We will look at Jackson's Hotel Lock problem in more detail, in particular the following paragraph:

“The idea is that the hotel issues a new key to the next occupant, which recodes the lock, so that previous keys will no longer work. The lock is a simple, stand-alone unit (usually battery-powered), with a memory holding the current key combination. A hardware device, such as a feedback shift register, generates a sequence of pseudorandom numbers. The lock is opened either by the current key combination, or by its successor; if a key with the successor is inserted, the successor is made to be the current combination, so that the old combination will no longer be accepted.” [1]

In the language of “Natural Deduction” this translates to the following statement :

(IF (SuccessorCombination=KeyCombination) THEN lock=open and LockCombination = SuccessorCombination) OR (IF (LockCombination=KeyCombination) THEN lock=open)

The bracketing used in this statement provides control and ensures that a “Vicious Circle” is avoided. Using the “Natural Deduction” element of Z only we can define the business rules for this solution. While the description is good it is still not precise enough and it would be impossible to automatically code from the description. It therefore needs to be converted into “Natural Deduction” so that the precise rules can be agreed upon and understood. A programmer could code directly from the description but that would require the business owner to read the code to double check that it had been interpreted correctly. The other option could be to test the resulting code to see if it behaves as expected. The issue here is that the number of combinations that would need to be tested is such a high number that in reality after checking a few routes statistically through the code, it is assumed to be acceptable.

4.1.3. James Martin's Model

In figure 2 the “Natural Deduction” statement can be displayed in a control tree in the form discussed by Martin [4].

Hotel Locking Model

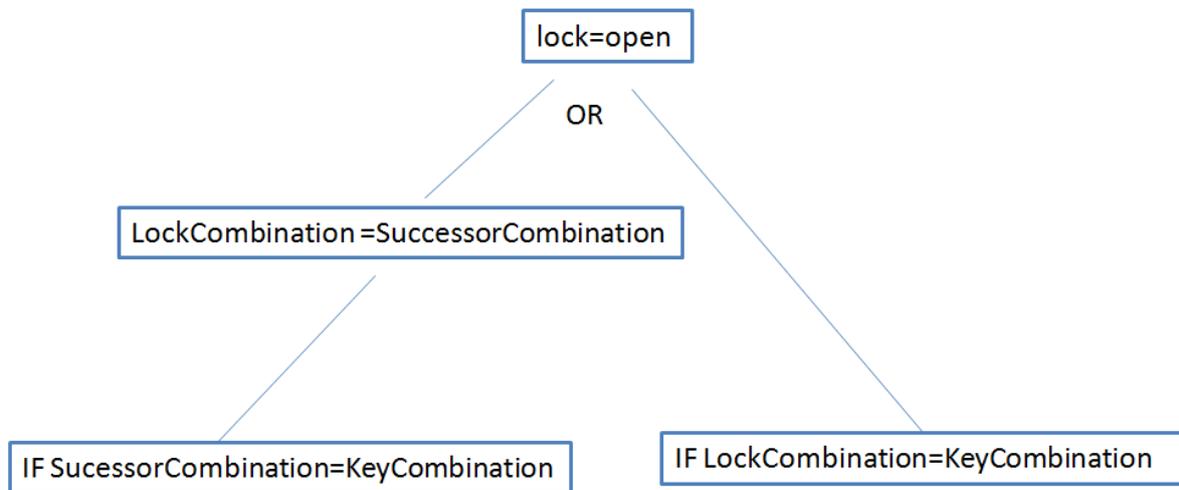


Figure 2 – The “Natural Deduction” statements shown as Martin’s control tree.[4]

4.1.3. UML

In Figure 3 there is a UML diagram, implemented in the tool EA, however it is noticeable that there are so many options in UML and EA that it is difficult to ensure that a clear meaning has been conveyed, as opposed to “Natural Deduction”. While simple logic can be displayed in one UML diagram, multiple diagrams are required for more complex problems, which increases the risk of misinterpretation.

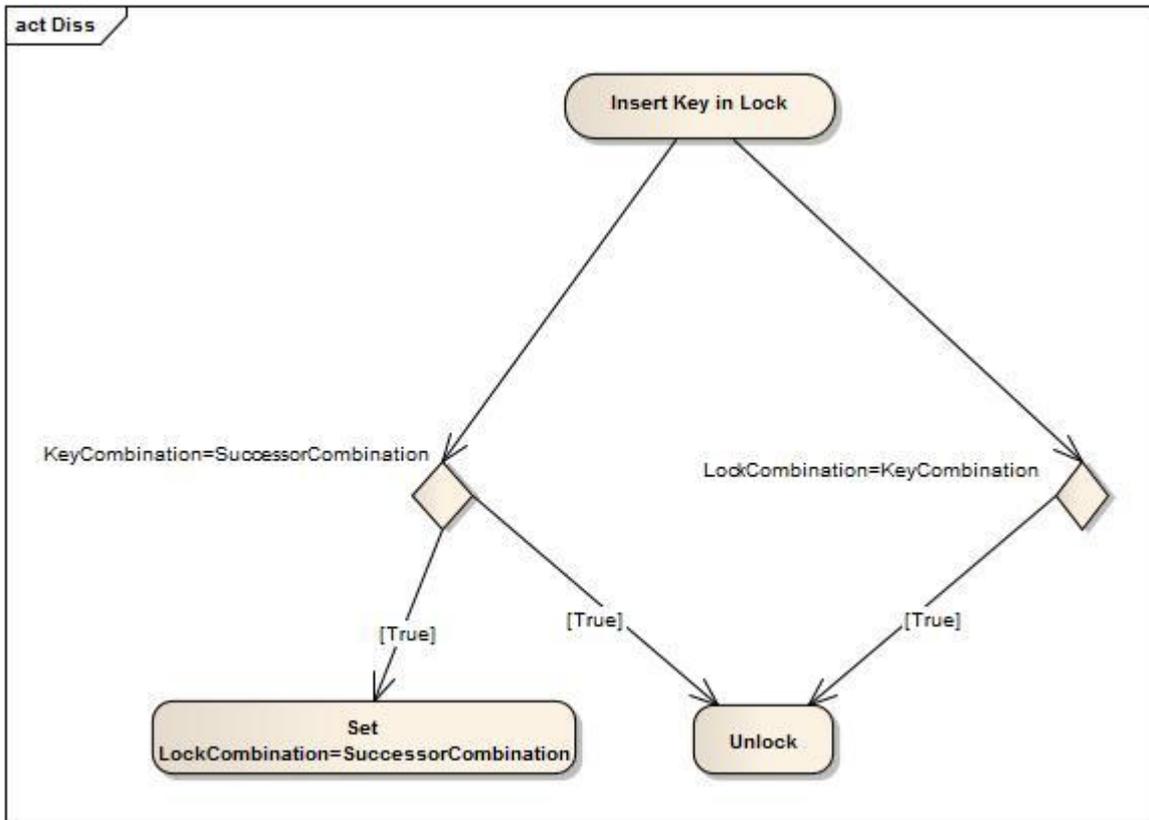


Figure 3: UML diagram of the Hotel Lock problem.

4.1.4. Alloy

In figure 4 the model provided by Jackson for the Hotel Locking problem has been executed.

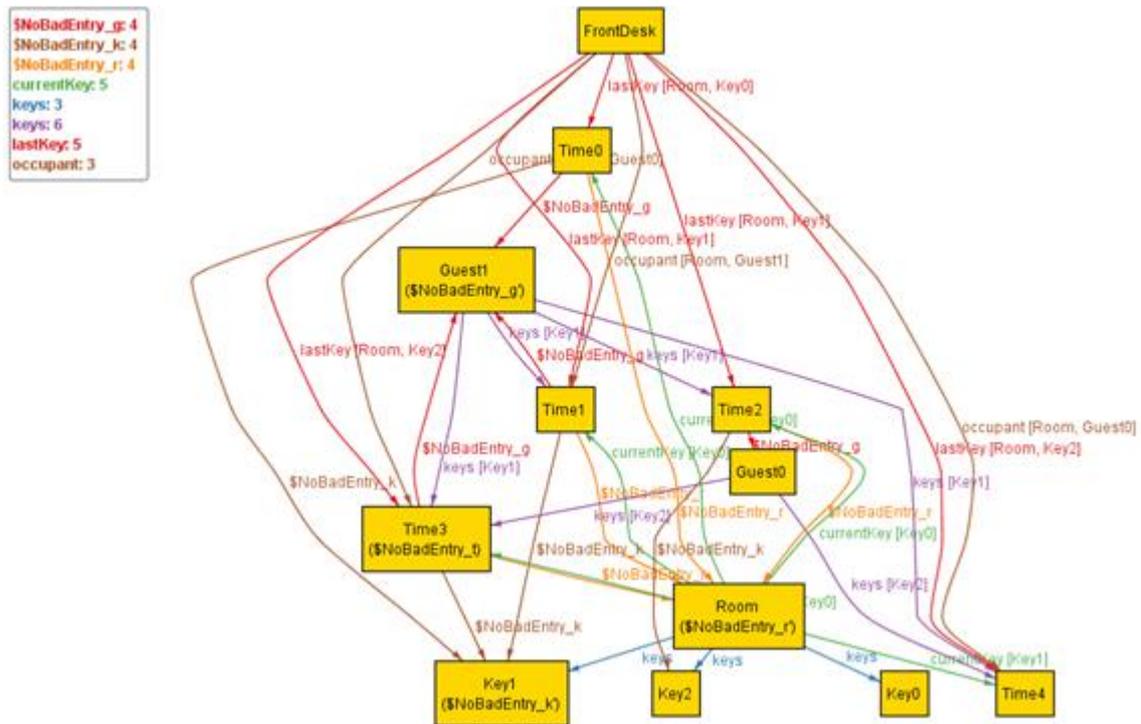


Figure 4 : Alloy model of the Hotel Locking problem.

5 Conclusions and Recommendation

The Alloy model provides an analysis of the paths through the Hotel Locking Problem using a SAT solver and will still only check a limited number of possible scenarios. The pictorial representation is useful but still not precise enough to code from. The Alloy language is almost as complicated as C so the user community may struggle to confirm that the Alloy text matches their problem.

The UML diagram approach is very popular with Software Engineers since it enables them to convey complex issues in simple diagrams. However the diagram cannot easily be converted into application code so it needs to be interpreted by the programmer and as it is a diagram it inevitably makes broad assumptions and lacks clarity in important areas.

While working on this dissertation I wrote an initial version of the statement in “Natural Deduction” and showed it to a colleague. After reading Jackson’s paragraph he found an error in my “Natural Deduction” statement and corrected it. This person, who cannot write programs and therefore may represent the user community, was able to check and correct a “Natural Deduction” statement based on the description. Another colleague, subsequently spotted in the “Natural Deduction” statement that I had missed the “successor” concept. Both colleagues pointed out the problem in the “Natural Deduction” statement and did not mention that the same issues were also in the UML and James Martin diagrams. I proceeded to correct my “Natural Deduction” statement and discovered two further errors, which would have been identified by a parser. A further colleague a very experienced programmer subsequently stated “Nothing is said about updating SuccessorCombination to its next value. Should something be said?”.

I have drafted the following which is my first attempt at the rules for the “Natural Deduction” parser written in “Natural Deduction”.

```
(If ( (logicaloperator="or") or (logicaloperator="and") or
(logicaloperator="not") or (logicaloperator="if" and
logicaloperator="then") ) then logicaloperator=ok) and ( if(
logicaloperator="(" and logicaloperator=")" ) then
logicaloperator=ok ) and ( if ( logicaloperator = "=" ) then
logicaloperator = ok)
```

My colleague also pointed out for the above statement “(logicaloperator="if" and logicaloperator="then") . . . ' is this conjunction possible?” I have started to wonder whether this illustrates the difficulty of interpreting requirements from English text or comprehension of Natural Deduction, more research is clearly required to establish this issue,

Tim Berners-Lee observed in “The Semantic Web as a language of logic” the following rules for a knowledge system:

“[...]a knowledge representation system must have the following properties:

1. *It must have a reasonably compact syntax.*
2. *It must have a well defined semantics so that one can say precisely what is being represented.*
3. *It must have sufficient expressive power to represent human knowledge.*
4. *It must have an efficient, powerful, and understandable reasoning mechanism*
5. *It must be usable to build large knowledge bases.*

It has proved difficult, however, to achieve the third and fourth properties simultaneously. “[11]

I would further add that when read by both a human and computer; the interpretation must be identical and decidable. I therefore propose that using restricted “Natural Deduction” and then applying the “Axiom of Reducibility” achieves these features and are therefore arguably the first law of Software Engineering.

6. Acknowledgements

I would like to thank the following for taking the time to give me the benefit of their wisdom, encouragement and criticism in private conversations regarding previous versions and sections of this paper for which I am very grateful: Professor Daniel Jackson of MIT ; Professor Sir Tony Hoare of Kellogg College, Oxford University; Professor Wilfried Sieg of Carnegie Mellon University; My tutor Dr. Ivan Flechais of Oxford University; My Professor of Software Engineering, Jim Davies of Kellogg College, Oxford University; Professor Jonathan Michie of Kellogg College, Oxford University; My Safety Critical Systems lecturer for his inspiration Dr Robert Collins of Kellogg College, Oxford University; Dr. Andrew Simpson of Kellogg College, Oxford University; Margaret Hamilton; Nick Watkinson; Dr. Wojtek Rappak; Alan Jones; Nick Bishop; Fraser Houchen and Phil Allen. To Kellogg College, Oxford University for giving me this opportunity I will donate 50% of any revenue of Patent 13103051 for further research of this subject.

7. References

1. Software Abstractions Logic, Language, and Analysis. Daniel Jackson. The MIT Press. 978-0-262-101141-1. (<http://softwareabstractions.org/sample-chapters/appendix.pdf>)
2. The Open Group Architecture Framework TOGAF 2007 edition (Incorporating 8.1.1). The Open Group. Van Haren Publishing. ISBN-978-90-8753-094-5.
3. Proof and Disproof in Formal Logic: An Introduction to Programmers, Richard Bornat, Oxford University Press. ISBN 0-19-853027-7.

4. Using Z, Specification, Refinement and Proof. Jim Woodcock and Jim Davies. Pearson. ISBN 9-780139-484728
5. System Design from Provably Correct Constructs. James Martin, Prentice-Hall. ISBN 0-13-881483-X
6. Collected Works of A.M.Turing – Mathematical Logic. R.O.Gandy & C.E.M. Yates editors. North-Holland. ISBN 0-444-50423-0
7. Software Requirements Objects, Functions, & States – Alan. M. Davis. Prentice Hall. ISBN 0-13-805763-X
- 8 The Logical Paradox – Charles Lutwidge Dodgson, Christ Church College, Oxford (Lewis Carroll) <http://www.jstor.org/pss/2250662> (Accessed 9/5/2011)
- 9 Principia Mathematica – Bertrand Russell and A.N. Whitehead.
10. TOGAF Version 9 Pocket Guide, The Open Group. Document Number: G092
11. The Semantic Web as a language of logic <http://www.w3.org/DesignIssues/Logic.html> (Accessed 15/05/2011)
12. An Investigation of the Laws of Thought, George Boole, Cambridge University Press ISBN 978-1-108-00153-3
13. Logic in Computer Science Modelling and Reasoning about Systems" by Michael Huth and Mark Ryan ISBN 978-0-521-54310-1
16. Software Engineering Concepts. Richard Fairley. McGraw Hill. ISBN 07-Y66272-X.
17. The Annotated Turing, Charles Petzold. Wiley. ISBN 978-0-470-22905-7.
18. The Principles of Mathematics. Bertrand Russell. Forgotten Books. ISBN 978-144005416-7.
19. Safety Critical Computer Systems. Neil Storey. Pearson. ISBN 0-201-42787-7.
20. The Computer and the Brain. John Von Neumann. Yale University Press. ISBN 9-780300-084733.
21. Principles of Mathematical Logic. D. Hilbert and W.Ackerman. AMS. ISBN 978-0-8218-2024-7
22. Higher Order Software Techniques – Applied to a Space Shuttle Prototype Program. Margaret Hamilton and Saydean Zeldin. The Charles Stark Draper Laboratory, Cambridge, Massachusetts. USA.

7. Appendix

7.1 Appendix I – Patent 1

USA Patent Application Number: 13103051

Filed: 07-MAY-2011

Title: COMPUTERIZED SYSTEM AND METHOD FOR VERIFYING COMPUTER OPERATIONS

V0.1 –28th October 2010- Paul William Watkinson

1. What is the title of your invention?

“System and Method for Verifiable Computing”

2. What are the objectives of your invention?

2.1. What problems does the invention solve?

Currently computing enables programs to be written and run that are not verifiable and provably correct. This dates back to Alonso Church’s papers “An Unsolvable Problem of Elementary number Theory” and “A Note on the Entscheidungsproblem” which were the basis for Alan M. Turing’s landmark paper “On Computable Numbers, with an Application to the Entscheidungsproblem”. Turing in this paper stated that “the terms of U must include the necessary parts of the Peano axioms...”.

2.2. How does your invention solve the problem or problems?

To date computers have been implemented that do not satisfy the “Peano axioms” and the “The axiom of Reducibility” as a result in Turing’s words it is possible to produce a program which can be executed which is not a “computable sequence”.

To resolve this problem it is necessary to enforce “Peano’s axioms” at the hardware level rather than at the software level. It is also necessary, and this was missed by Turing, to enforce A.N. Whitehead and B. Russell’s “The axiom of Reducibility” from their “Principia Mathematica – Volume One” to ensure that the functional decomposition is also hierarchical.

2.3. How does your invention differ from already patented or made inventions?

To achieve this the hardware solution must only allow three primitives as defined by John Von Neumann in his book “The Computer and the Brain” thus stated “”And” and “Or” are the basic operations of logic. Together with “No” (the logical operation of negation) they are a complete set of basic logical operations, no matter how complex, can be obtained by suitable combinations of these”.

The hardware solution must also read in the entire “computable sequence” before execution to check that the functional decomposition is “hierarchical” from “The Axiom of Reducibility” and satisfies the “Peano Axioms”.

The solution therefore ensures that only programs are executed that are provably correct.

3. What are the uses of your invention?

The invention will enable software programs to be trusted as provably correct because the resulting computer can only run programs that satisfy this requirement.

4. List the parts of your invention or steps in the process and how the parts or steps relate to each other.

(i) Logic Component: This component will only execute John Von Neumann's three primitives "and", "or" and "no".

(ii) Control Component: The control component will read the entire "sequence of functions" or "the program" before execution to verify that the "functional decomposition is 'hierarchical'" and satisfies the "Peano axioms" and The Axiom of Reducibility". The control component will then execute the "computable sequence" against the "primitives" in the "logic component".

(iii) Primitives Component: The primitives "and", "or" and "no" represent the lowest element of the computer stack.

(iv) Hierarchy Component: Before the "program" or "computable sequence" is executed the "Control" component will request that the functional decomposition is "hierarchical" as defined by A.N. Whitehead and B. Russell in "Principia Mathematica Vol1 (1910) – The Hierarchy of Functions" and leads from B.Russell's book "An Essay on the Foundations of Geometry (1897)" which states "Being a creature of the intellect, can be dealt with by pure mathematics". It is also the result of Paul William Watkinson's Master's thesis for Software Engineering at Kellogg College, Oxford University (2010) "Software Engineering – Methodology for Critical Systems".

7.2 Appendix 2 – Patent 2

USA Patent Application Number (Provisional): 61484377

Filed: 10-MAY-2011

Title: System And Method For Describing The Logic Of Software Requirements

System and Method for Describing the Logic of Software Requirements.
V0.5 – 10/5/2011

Patent owners:

1. Paul Watkinson
2. Alan Jones,
3. Wojtek Rappak.

1. What is the title of your invention?

System and Method for Describing the Logic of Software Requirements.

2. What are the objectives of your invention?

To capture the logical structure of requirements through a notation which is concerned only with their universal format and logical rules.

2.1. What problems does the invention solve?

Clarification and logical verification of requirements at an earlier stage of the development cycle.

2.2. How does your invention solve the problem or problems?

It uses "Natural Deduction" to describe the requirements so they are precise and can be interpreted by a human or computer in the same way. "Natural Deduction" in this interpretation consists of only four connectives, "and", "if-then", "or" and "not", or in latinized that would be "Conjunction", "Implication", "Disjunction" and "Negation", to give a precise definition.

2.3. How does your invention differ from already patented or made inventions?

It provides a tighter link between a logically correct requirement description that can be understood by a human and a specification that can be transformed into a computer process. The "method" of applying "Natural Deduction" to the capture of requirements ensures that the user can "only" describe the requirement in terms of the "Natural Deduction" logic. Once this logical statement has been captured then it can be displayed in a number of formats, including but not exclusively: text, Z, UML, C, Java etc.

3. What are the uses of your invention?

To make it easier, faster and more accurate to define software requirements through a software tool which ensures that their description is logically correct.

4. List the parts of your invention or steps in the process and how the parts or steps relate to each other.

- a) Logic component that uses "Natural Deduction".
- b) Input component that enables the requirements to be captured.
- c) Presentation component to enable the requirements to be displayed.
- d) The input component will use the Logic component (a) to ensure that the requirements are precisely described.