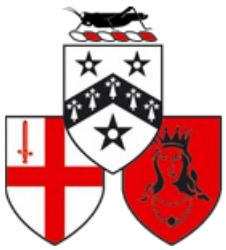

GRESHAM COLLEGE

Founded 1597

Making Software 'Correct by Construction' Transcript

Date: Tuesday, 2 May 2017 - 6:00PM

Location: Museum of London

2 May 2017

Making Software ‘Correct by Construction’

Professor Martyn Thomas

Introduction

We have seen in previous lectures that the common approach to writing software (write it, test it and fix the defects that you find) results in software that still contains a large number of errors[i].

This lecture explores an alternative approach where you start by writing a requirements specification in a way that makes it possible to analyse whether it contains logical omissions or contradictions. Then you develop the software in a way that provides very strong evidence that the program implements the specification (and does nothing else) and that it will not fail at runtime. We call this making software “correct by construction”, because the way in which the software is constructed guarantees that it has these properties.

Proving that software meets its specifications has been a goal since the earliest days of computing, but it was long considered to be impractical except for the smallest programs – and excessively expensive even for these. Nevertheless, some of the greatest computer scientists and software engineers persisted because they could see that proofs of large programs were theoretically possible and that nothing else would deliver the very high assurance that would be needed[ii]. The combination of advances in computer science and enormous increases in computer processing speed and storage capacity has brought success. As I will show in this lecture, it is now possible to produce highly dependable software – and to prove that it is dependable – faster and cheaper than most companies produce averagely buggy software using traditional software development methods. I will show that some companies are using these methods routinely in their businesses and even offering guarantees for their software rather than lengthy end-user licence agreements (EULAs) that attempt to disclaim all possible liability if the software is unreliable and insecure.

Alan Turing: ‘Reasoning about a large routine’ in 1949

The earliest computer scientists realised that it would be necessary to reason about software because just testing it would be inadequate. Professor Cliff Jones[iii] has compared papers written in 1947 and 1949 by Herman Goldstine and John von Neumann[iv], Alan Turing[v], and Haskell Curry[vi]; he comments that

“Frustratingly these early insights appear to have then lain dormant only to be reinvented (and developed enormously) starting two decades later … It is Alan Turing's 1949 paper that has the clearest similarity with what followed. … The overall task of proving that a program satisfies its specification is, like most mathematical theorems, in need of decomposition. Turing made the point that checking the addition of a long series of numbers is a monolithic task that can be split into separate sub-tasks by recording the carry digits. His paper displays five (four-digit) numbers whose sum is to be computed and he makes the point that checking the four columns can be conducted separately if the carry digits are recorded (and a final addition including carries is another separable task). Thus the overall check can be decomposed into five independent tasks that could even be conducted in parallel. Turing's insight was that annotating the flow chart of a program with claims that should be true at each point in the execution can also break up the task of recording an argument that the complete program satisfies its specification (a claim written at the exit point of the flowchart). … There is [also] a delicate and important issue about loops that Turing addressed formally: the need to argue about their termination. … The fascinating thing about the 1949 paper was the early recognition of the need for something more mathematical than the execution of test cases to support the claim that a program satisfies its specification. Of course, the example given in a (very short) paper is small but it is clear from the title of the paper that the intention was to apply the proposal to `large routines'. Turing gave a workable method for recording such reasoning. … It should be clear that a major intellectual step had been made beyond the acceptance that `write then debug' was the only way to achieve correct software.

Turing’s influence on what followed cannot be explored in detail here; interested readers are strongly recommended to read Professor Jones’ paper *Turing and Software verification*[vii].

Automated Program Analysis

By the early 1970s, Hoare, Floyd, Dijkstra, Jones and others had provided the computer science foundations for program verification. Professor Bernie Cohen (City University) summarises the position as follows[viii]:

*A few mathematicians realised that the verification of computer programs would require something more analytical than testing and flowchart walk-throughs. Böhm and Jacopini showed that all 'imperative' programs (i.e. those executed by 'von Neumann' machines) could be represented in flowcharts containing only three constructs: sequence, selection and iteration**[ix]**. Floyd and Hoare showed that these constructs, together with the 'assignment' of values to variables, could be construed as predicate transforms. Their composition could be used in a formal proof that the program's operation would always satisfy some logical predicate ranging over the values of its input and output variables. Edsger W. Dijkstra designed a language (which did not include a GOTO statement) whose semantics were completely defined in this way, together with a logic of weakest preconditions in which to conduct the requisite proofs of correctness. He illustrated the technique in the pyrotechnical display of problem solving called A Discipline of Programming**[x]**.*

In my second lecture[xi] I mentioned the NATO software engineering conferences in 1968 and 1969 and showed that they had diagnosed the causes of the first software crisis. In the reports[xii] from these NATO conferences, E.S Lowry from IBM is quoted as saying

"Any significant advance in the programming art is sure to involve very extensive automated analyses of programs. … … Doing thorough analyses of programs is a big job. … It requires a programming language which is susceptible to analysis. I think other programming languages will head either to the junk pile or to the repair shop for overhaul, or they will not be effective tools for the production of large programs."

To make it practical to analyse large programs the analysis had to be automated; in the early 1970s this was the subject of a secret research project at the Ministry of Defence Royal Signals and Radar Establishment (RSRE) at Malvern in the UK. It had become clear that software would be of increasing importance to national defence and intelligence operations and that it would be essential to ensure that programs did not contain errors or 'trojan code' that leaked secret data. Automated analysis of control flow and information flow became an important challenge and methods of analysing control flow and data flow were developed by Bob Philips (Head of Command and Control Applications at RSRE) in collaboration with an academic computer scientist Bernard Carré. After Bob Philips' untimely death from a heart attack, Dr John Cullyer took over the research[xiii] and made it better known across Government. Dr Brian Gladman, Head of the Computer Division at RSRE, later declassified and released the research because he could see the importance that this would have for high-integrity software in industrial and commercial civil applications.

Two commercial software products resulted from this research: MALPAS, which is now supported by Atkins[xiv] and SPARK[1] which was developed By Bernard Carré (first at the University of Southampton and then in his company Program Validation Limited). PVL joined my own company, Praxis, in 1994 and SPARK has been further developed and supported by Altran UK[xv] which took over the Praxis Critical Systems Division in 1997. There is a free version of the SPARK 2014 toolset that can be downloaded from the Adacore Libre website[xvi] and GitHub[xvii].

Making Software Correct by Construction

The key to making software right first time is to have zero tolerance for errors. Errors are inevitable—to err is human —but errors should be avoided as much as possible, found as soon as possible after they are introduced and, if any errors survive to later stages in the project, then the development process should be improved to ensure that errors of that sort are found quickly next time and forever. Anthony Hall has written an excellent summary of the principles[xviii].

All programmers are familiar with automated tools that check programming languages for syntax errors: these depend on formal grammars: mathematically formal descriptions of the structure of the language and they carry out a simple form of *static analysis* (static, because the checking is done without executing the program, in contrast with *dynamic testing*). If the formal description of a language contains information about its meaning (*semantics*) in addition to its structure (*syntax*) then static analysis can reveal much deeper errors in a program. For example, if the language allows the programmer to state the limits on the value of integer variables (and the programmer does so), it becomes possible to check statically the maximum and minimum values of all integer expressions and to detect the possibility of division by zero, arithmetic overflow and violation of array bounds.

These ideas have been developed much further, with mathematically defined languages that allow software engineers to specify the intended behaviour of their programs in detail, so that as the design is developed and more detail is added, static analysis can identify inconsistencies that reveal errors. There are several such *formal methods* that are in use in industry for developing software with high assurance that the result is correct. Leading examples are Méthode B[xix] with the Rodin toolset[xx] (that has been qualified by Siemens Transportation for use in safety-critical applications), and SPARK[xxi], which has been used by Altran and several

other companies for many safety-critical and security-critical applications. I shall draw heavily on SPARK and the methods used by Altran in this lecture.

What does it mean for software to be “correct”? There are three main criteria

- It must carry out the functions that are required and do nothing else
- It must continue to function as required for all possible inputs
- There must be very strong evidence that these properties are true and the evidence for this must be independently verifiable.

The first criterion implies that there must be an unambiguous statement of all the required functions; the second that the software must not fail through memory leaks, address violations, accessing null pointers, arithmetic overflows or the many other creative ways that programmers have found to cause their systems to crash. The third criterion means that each stage in the software development must also create the evidence needed for assurance, otherwise the costs would make it impractical to generate sufficient evidence.

All software projects start with an informal statement of requirements. With *agile* methods this is typically a set of “user stories” that describe a number of things that various users may wish the system to do and how the system should respond in each case. Traditional projects often start with a list of requirements that state that the system *should* do this, or *should* do that – there can be hundreds of such statements. Almost always, these requirements will be in English or some other natural language, usually supplemented by a few diagrams. Except in trivially simple examples, such statements of requirements are *always* incomplete and contradictory because natural languages and diagrams cannot express and communicate complex ideas consistently and completely. (A clear demonstration of this is provided by the tax laws in every country, despite—or possibly because of—the efforts of many top legal brains over many years). The consequence for software is that despite the specifications, designs, user manuals, on-line help systems and other documentation, the ways in which most programs behave and the ways in which they can be led to misbehave are a continuing source of surprise and dismay to users and software developers alike.

Correct-by-construction methods must therefore start by expressing these informal specifications in a language that is unambiguous and that can be analysed to expose any contradictions. Mathematics provides the tools for doing this and for most purposes elementary mathematics is all that is needed. Computer scientists have invented several of these rigorous notations (*formal methods*) that have strong mathematical foundations and that are designed to make the task of specifying the required behaviour of software as practical as possible. Méthode B[xxii], VDM[xxiii], Alloy[xxiv], Lustre/SCADE[xxv] and Z[xxvi] are five that are quite well known and widely used.

In the projects that I shall describe, Altran used Z as their specification language.

The TOKENEER Demonstrator for the US National Security Agency (NSA)

In 2002, the NSA approached Altran UK with a challenge (at this time the company was called Praxis Critical Systems but I shall call them Altran or Altran/Praxis).

The international IT security community had laboriously agreed an international software security standard in which IT products are certified against standard security requirements, called the Common Criteria for Information Technology Security Evaluation (*Common Criteria*). This standard (ISO/IEC 15408[xxvii]) includes Evaluation Assurance Levels (EAL1 to EAL7) that define the degree of assurance that a product meets its security requirements (called its *Protection Profile*). The seven levels are

EAL1: Functionally Tested

EAL2: Structurally Tested

EAL3: Methodically Tested and Checked

EAL4: Methodically Designed, Tested and Reviewed

EAL5 Semiformally Designed and Tested

EAL6 Semiformally Verified Design and Tested

EAL7 Formally Verified Design and Tested

The NSA's problem was that suppliers were claiming that EAL5 and above were too expensive and too difficult, so Altran were asked to develop a demonstrator system under the supervision of the NSA to explore whether their Correct-by-Construction methods were a practical and cost-effective way to achieve EAL5.

The NSA have a demonstration security system called *Tokeneer* that they use for trials of new technologies, comprising a physically secure enclave that provides protection to secure information held on a network of workstations. Access to the enclave is controlled by a security door under the control of an ID Station that reads encrypted biometric information from a user's security pass, compares it with a fingerprint from the user and if a successful identification is made and the user has sufficient clearance (also held on the pass), the lock on the enclave door is released to allow the user access to the enclave.

The NSA asked Altran to redevelop the software for the Tokeneer ID Station (TIS) to meet EAL5 using their correct-by-construction methods and to deliver the software and all the project documentation, skills profiles and metrics so that the formal development could be compared with previous non-formal development of the system[xxviii].

The project manager and technical leader of the Altran team was Janet Barnes. Altran re-developed the Core functions of the Tokeneer ID Station (TIS)

- Controlling User access to the enclave
- Checking the pass token and biometrics, and unlocking the door
- Creating an audit trail of all activity
- Monitoring the door state and raising alarms if the door is left open
- Plus various administrative functions
- Guard -Override Door
- Security Officer –Shutdown or Change Configuration
- Audit Manager –Archive Log

Software was written to simulate all the Tokeneer peripherals.

The first step was to agree the two system boundaries of interest; the boundary between the ID Station machine and its environment (including its peripherals); and the boundary between the ID Station core functions and its support functions.

The Correct by Construction process is designed to validate each lifecycle phase as early as possible and to verify that each successive phase is consistent with the phase that precedes it, so that faults are detected and eliminated as quickly and cheaply as possible. The project report[xxix] shows the development process …

… and the assurance process.

This process was designed to be as formal as possible whilst remaining practical. **Requirements** were captured and documented using a process based on Michael Jackson's *Problem Frames* ***[xxx]*** and reviewed and agreed with the NSA. The **formal specification** was then written in Z supplemented with an English description. The project report explains that

"the Z notation uses data types and predicate logic to describe the way in which the system will behave; Z is particularly powerful because of its use of schemas to decompose the specification into small components that can be reasoned about individually and then combined to describe the system as a whole. The Z notation provides an unambiguous specification language while the English narrative assists readers, writers and reviewers in understanding the intention of the Z by providing a secondary description of the system and advice as to how to interpret the Z."

The formal specification defines the *states* that the system can be in (for example, the door may be open or closed) and the *operations* that can change those states. It can be shown to be complete and self-consistent and it provides an unambiguous description that can be used to check that the security requirements (derived from the required protection profile and also expressed in Z) and the subsequent design are consistent with the functional specification.

Altran then produced a formal design. The project report describes this step as follows:

The aim of the formal design is to elaborate the abstract aspects of the Formal Specification to explain how the system will be implemented. The Formal Design describes the system in terms of concrete state and operations using types that are easily implemented. The Formal Design is the source of required functional behaviour used during implementation.

The Formal Design was written using the Z notation accompanied by English narrative. There were a number of ways in which we developed the abstract specification to a concrete design.

- *The Formal Design elaborated those aspects of the Formal Specification where there was insufficient detail to move directly to implementation. For example, the Formal Design describes the contents of the audit log and describes how the log should be implemented in terms of local files.*
- *The Formal Design elaborated aspects of the real world which had been left slightly abstract. Abstractly it is sufficient to know that a certificate is validated using a key — this is refined in the design to describe a certificate as a portion of raw data and a signature with an algorithmic relationship between the signature and the data dependant on a key.*
- *The Formal Design removed non-determinism from the system. Where more than one operation could proceed in the specification additional pre-conditions were added to prioritise the operations. For example, logging-out an administrator was given a higher priority than continuing with a long-lived user entry operation.*
- *The Formal Design restructured some operations to reduce the step to implementation, for example the action of logging-out an administrator was removed from all other operations and considered separately as it would take priority in the design.*

The Formal Design was written using the same notation as the Formal Specification as this provides benefits of reuse. Where the level of detail in the specification is sufficient for the design, data types and state schemas can be carried forward unchanged. By using the same notation it is clear where the design has introduced refinement. Moreover, it is possible to demonstrate that the refinement is valid by defining a retrieve relation that relates the concrete and abstract versions of the state and proving a number of relationships between the abstract and concrete versions of the operations. On TIS this activity was limited to those operations where the refinement relation was non-trivial, for example, adding elements to the log.

Refinement proofs can be done for all operations, and this can be a powerful technique to uncover design errors before implementation starts. Proof can be carried out rigorously, but by hand, or can be carried out using proof tools. In practice, the discipline of writing the retrieve relation ***[xxxi]*** *and carrying out some sample proofs can uncover the majority of errors.*

The Altran team then designed the software architecture, defining the way in which the system functions would be distributed between separate software components and linking the formal specification in Z to the software structure of SPARK Ada packages. Altran call this the INFORMED (*Information Flow Oriented Method of object Design*) design. The package descriptions were defined in SPARK Ada and the package bodies were written, also in SPARK[xxxii].

SPARK is a strict subset of Ada (the latest version complies with Ada 2012) with formal annotations (*contracts*) that contain the specification. In the version of SPARK used for Tokeneer, these contracts had to be expressed as Ada comments so that programs in SPARK[xxxiii] could be compiled by any Ada compiler, removing the need for dedicated SPARK compilers and allowing the use of different compilers and portability to different computer hardware, but after the time of Tokeneer, Ada 2012 became available and included a feature that allows contracts to be written directly in Ada.

The SPARK toolset comprises a suite of tools that check conformance of the program with the language rules and with the embedded contracts and that provide the mathematically rigorous assurance that the program is correct by proving the **verification conditions** that are generated or by explaining how the program could fail.

The system was tested against the expected (specified) behaviour to check for any errors introduced due to erroneous interaction between modules within the system. Finally, the software was sent to the NSA's independent assurance team (SPRE) in Albuquerque, New Mexico, who reported that the software contained zero defects.

A report was delivered as part of the project, giving all the productivity and size metrics for each phase.

The NSA were startled by the productivity figures because they were much higher than had ever been achieved on an NSA project despite having gone beyond the minimum requirements for Evaluation Assurance Level EAL5; Altran's C by C development process includes activities from the much more rigorous EAL6 and EAL7 levels.

The NSA wondered whether this remarkable performance had only been achieved because Altran had used an expert team that was very familiar with the process, languages and tools; they therefore decided to add a second phase to the experiment, by asking three students to add new functionality to the software that Altran had delivered. The students (two undergraduate students studying mathematics and computer science and one

computer science graduate student) had no previous experience of Z and only one of them had ever experienced SPARK. They were given twelve weeks to adapt the Praxis code to run on the real hardware system rather than interfacing to the simulated peripherals, which involved changing the Ada & SPARK code with help from SPRE. They were then asked to add new functionality, following the entire correct-by-construction methodology to add a keypad device and a requirement for a password.

The students were given 3-4 days training on reading and writing Z, 3 days training on the Tokeneer ID Station, 2 days training on Ada and 4-5 days training on SPARK and the SPARK tools. Beyond that they only had Z and SPARK textbooks and manuals, and email support from Altran/Praxis if they needed it.

In the allotted 12 weeks, the students added new functionality to

- the requirements document
- the functional specification in Z
- the design document in Z, and
- the SPARK code and annotations

They ran the SPARK tools to analyse and prove the modified software, and tested it. No defects were found in the student's software.

The NSA's conclusions were reported to a SPARK User Group meeting by Randolph Johnson of the NSA. He stated that

The Correct by Construction (C by C) process

- *meets Common Criteria and ITSEC security requirements for EAL5*
- *produces code more quickly and reliably and at lower cost than traditional methods*
- *has a reasonable learning curve*
- ***C by C is proven and practical***

With some difficulty, the Altran Tokeneer team persuaded the NSA to make the entire project available to the international research community so that others could reproduce it, experiment by introducing errors and seeing where they were found, trying alternative proof strategies and formal verifiers, or by re-implementing the system in other languages. In 2008 the NSA finally agreed and all the project deliverables—the requirements, specifications, security policy, formal specifications, code, proofs, tests, results, metrics and reports—are still available[xxxiv] for free download, along with free copies of the SPARK toolset.

In the following years, a small number of errors have been found in the Tokeneer software by academic researchers[xxxv]. The authors of the cited paper say *It is a remarkable achievement that this complete redevelopment of an actual system by 3 people over 9 months (part-time) achieved such a level of quality that only a few problems can be found several years later using so many new approaches not available at the time.*

In 2011, Professor Sir Tony Hoare presented the inaugural *Microsoft Research Verified Software Milestone Award* to Janet Barnes and Roderick Chapman for the Tokeneer project. Professor Hoare said "*Congratulations to Janet and Rod as well-deserved recipients of this award. And thanks to Altran Praxis and the US National Security Agency for their commitment to their project. It has given a persuasive demonstration of the cost effectiveness of formal methods in application to security software, and complements similar experience at Microsoft".*

Correct by Construction methods have been used on several important projects and some of Altran's customers have agreed that details can be made public.

Correct-by-Construction Projects (1). *Smart Card Security*

Correct by Construction methods were used to build the back-end security infrastructure and cryptology for the Certification Authority for the MULTOS smart card[xxxvi]. The security assurance required was ITSEC E6.

The specification was written in Z and Z type checking was performed. Then the code was designed and written in SPARK. The required security properties translated to SPARK specifications and the code was proved to maintain the security properties, using the SPARK tools.

The project was 100,000 lines of SPARK, Ada, C, C++ and SQL. Three trivial defects and one specification defect were reported and fixed under warranty in the first year, a defect density of 0.04 defects per thousand lines of

code (KLoC).

Correct-by-Construction Projects (2). *Air Traffic Control*

Correct by Construction methods were used to build a new suite of air traffic management tools for the UK National Air Traffic Services NATS.

The iFACTS system enables UK air traffic controllers to handle more flights safely, by providing electronic flight-strip management, medium-term conflict detection and trajectory prediction fifteen minutes ahead and other monitoring tools. It has been in full operational use since December 2011, handling every climb, turn and descent in UK airspace[xxxvii]. The diagrams below of the UK airspace and an iFACTS screen are reproduced with the permission of Neil White, Head of Engineering at Altran UK.

The following description of the iFACTS project is based on a keynote talk *Are We There Yet? 20 Years of Industrial Theorem Proving with SPARK* presented by Roderick Chapman and Florian Schanda at ITP 2014, the fifth conference on Interactive Theorem Proving and related issues[xxxviii].

iFACTS has a formal functional specification (mainly in Z) and 529k lines of executable SPARK code (250k logical lines of code, as counted by GNATMetric). The proof focused on type-safety rather than functional correctness because of the need to demonstrate stringent requirements for reliability and availability. The system produces over 150,000 verification conditions, of which almost 99% are proved entirely automatically by the SPARK Simplifier. A full proof run takes about 3 hours on a fast desktop computer but this has been reduced to 15 minutes by adding a proof caching system that checks whether the proof is already known. This rapid turnaround means that developers can quickly check that their changes have not broken the proof, meaning that any errors are detected almost immediately.

Other Correct By Construction Projects

Other Altran Z/SPARK projects are described in the referenced paper by Roderick Chapman and Florian Schanda, but Altran are not the only company using SPARK, nor is SPARK the only technology for building software that is Correct by Construction. Event-B[xxxix] is increasingly widely used, for example, and the experiences of a number of companies with Event-B have been published[xl]

Conclusions

The use of mathematically formal software development methods is now practical and cost-effective in a growing range of applications. Correct by Construction methods make it possible to guarantee software and to provide high assurance for the safety and cybersecurity of the many systems for which these properties are vital. Software development may at last be emerging from its craft stage and becoming a mature engineering discipline.

Acknowledgements

I would like to thank Janet Barnes, Roderick Chapman, Allan Fox, Brian Gladman and Neil White for their assistance in preparing this lecture. All remaining errors are mine.

References

[1] *SPARK is completely unrelated to Apache SPARK: an open-source cluster-computing framework.*

[i] See in particular: https://www.gresham.ac.uk/lectures-and-events/should-we-trust-computers,

https://www.gresham.ac.uk/lectures-and-events/how-can-software-be-so-hard,

https://www.gresham.ac.uk/lectures-and-events/cybersecurity, and https://www.gresham.ac.uk/lectures-and-events/safety-critical-systems.

[ii] https://www.adelard.com/assets/files/docs/issre02_34_bishop.pdf and the referenced papers.

[iii] *Turing's 1949 Paper in Context,* Cliff B Jones, Newcastle University

[iv] Herman H. Goldstine and John von Neuman. Planning and coding of problems for an electronic computing instrument. Technical report, Institute of Advanced Studies, Princeton, 1947.

[v] A. M. Turing. Checking a large routine. In Report of a Conference on High Speed Automatic Calculating Machines, pages 67-69. University Mathematical Laboratory, Cambridge, June 1949.

[vi] Haskell B Curry. On the composition of programs for automatic computing. Naval Ordnance Laboratory Memorandum, 9806(52):19-8, 1949.

[vii] C B Jones, *Turing and Software verification,* CS-TR-1441 December, 2014, which is available online at

http://www.cs.ncl.ac.uk/publications/trs/papers/1441.pdf

[viii] B Cohen, A *Brief History of 'Formal Methods*, available online at http://www.drisq.com/Portals/0/Docs/A%20Brief%20History%20of%20Formal%20Methods.pdf

[ix] Böhm, C., Jacopini, G.: F*low diagrams, Turing machines and languages with only two formation rules.* Communications of the ACM, pp. 366–371 (May 1966)

[x] https://www.amazon.co.uk/Discipline-Programming-Automatic-Computation/dp/013215871X

[xi] https://www.gresham.ac.uk/lectures-and-events/a-very-brief-history-of-computing-1948-2015

[xii] http://homepages.cs.ncl.ac.uk/brian.randell/NATO/index.html

[xiii] It is likely that Ian Currie and Mike Foster (leading computer scientists at RSRE) contributed to the research.

[xiv] http://malpas-global.com/

[xv] http://www.spark-2014.org/about

[xvi] http://libre.adacore.com/

[xvii] https://github.com/AdaCore/spark2014

[xviii] http://www.anthonyhall.org/html/principles.html

[xix] http://www.methode-b.com/en/accueil/

[xx] http://www.methode-b.com/en/download-b-tools/rodin/

[xxi] http://www.spark-2014.org/

[xxii] http://www.methode-b.com/en/accueil/

[xxiii] C. B. Jones. Systematic Software Development using VDM. Prentice Hall International, Second edition, 1990.

[xxiv] http://alloy.mit.edu/alloy/

[xxv] http://www.esterel-technologies.com/about-us/scientific-historic-background/

[xxvi] http://spivey.oriel.ox.ac.uk/mike/zrm/

[xxvii] http://www.commoncriteriaportal.org/cc/

[xxviii] http://www.adacore.com/uploads/downloads/Tokeneer_Report.pdf section 1.1.2.

[xxix] Ibid.

[xxx] https://www.amazon.co.uk/Problem-Frames-Analysing-Structuring-Development/dp/020159627X

[xxxi] *Retrieve relation* is the commonly used term for the relationship between concrete and abstract states

[xxxii] John Barnes, *SPARK, the Proven Approach to high integrity software*, Altran Praxis 2012. ISBN 978-0-9572905-0-1. https://www.amazon.co.uk/d/Books/Spark-Proven-Approach-High-Integrity-Software-Barnes/0957290500/

[xxxiii] http://www.cambridge.org/ge/academic/subjects/computer-science/programming-languages-and-applied-logic/building-high-integrity-applications-spark?format=PB#bookPeople

[xxxiv] http://www.adacore.com/sparkpro/tokeneer/download/

[xxxv] Moy and Wallenburg, *Tokeneer: Beyond Formal Program Verification,* http://www.open-do.org/wp

content/uploads/2010/04/ERTS2010_final.pdf

[xxxvi] http://www.anthonyhall.org/html/papers_on_formal_methods.html#ieeesw

[xxxvii] Rolfe, M.: H*ow technology is transforming air traffic management.* http://nats.aero/blog/2013/07/how-technology-is-transforming-air-traffic-management

[xxxviii] http://proteancode.com/wp-content/uploads/2015/05/keynote.pdf

[xxxix] http://www.event-b.org/

[xl] *Industrial Deployment of System Engineering Methods*. Romanovsky and Thomas (Eds)

http://www.springer.com/gb/book/9783642331695#