Electronic Communications of the EASST
Volume 46 (2011)

Proceedings of the 11th International Workshop on Automated Verification of Critical Systems (AVoCS 2011)

Experiences in the Industrial use of Formal Methods

Janet Barnes

15 pages

Guest Editors: Jens Bendisposto, Cliff Jones, Michael Leuschel, Alexander Romanovsky
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Experiences in the Industrial use of Formal Methods

Janet Barnes

janet.barnes@altran-praxis.com, http://www.altran-praxis.com/
Altran Praxis Ltd, 20 Manvers Street, Bath, UK.

Abstract: Altran Praxis has used formal methods within its high integrity development approach, Correctness by Construction (CbyC), for a number of years. The Tokeneer ID Station (TIS) developed for the US National Security Agency (NSA) is one example of a development using formal methods and the CbyC approach. This project used a number of rigorous techniques including formalisation of the specification using the Z Notation, refinement of the specification to a formal design, software development in SPARK with proof of absence of run-time errors of the software and proof of system properties. The project has stood up well to the intense scrutiny it has been subject to since it became available to the wider community in 2008, with only five errors being found. Despite the general success of the approach there are challenges to using formal methods in an industrial context. By looking at a number of key properties that affect the success of deployment of tools and techniques in industry we attempt to put the challenges of industrial deployment of formal methods into perspective.

Keywords: Correctness by Construction, Formal Methods, SPARK, Tokeneer, Z

1 Introduction

The application of formal methods to the development of software has long been considered by industry as niche; only applicable to the development of core functions in particularly critical domains, where safety or security is paramount. Industry in general perceives the application of formal methods to be prohibitive for a number of reasons: cost, familiarity and maturity often being cited [Hal90].

Altran Praxis has applied formal methods in a number of its development projects [Hal96, HC02, KHCP00, TIS]. This paper looks at the way that Altran Praxis approaches software development via its Correctness by Construction approach [Ame06], considering how formal methods support the fundamental goals of the approach. It then explores the Tokeneer project as an example of a CbyC implementation where formal methods were adopted at every point in the lifecycle. Taking the view of an experienced industrial user of formal methods this paper takes a critical look at some of the criteria that impact the actual and perceived success of the adoption of formal methods. In conclusion, this paper questions whether industry is in a position to drop long held prejudices that Formal Methods are too challenging to use in practice and considers what changes are needed to fully overcome such prejudices.

2 Correctness by Construction

Over 20 years Altran Praxis has distilled the essence of best practice, captured from observation and experiences, into a principle of software development referred to as Correctness by Construction (CbyC). The key philosophy of CbyC is to avoid the introduction of errors; but where errors are injected, to find and remove them as early as possible; and to gather certification evidence efficiently as a natural by-product of the process.

2.1 Applying Correctness by Construction

Correctness by Construction does not prescribe particular tools or techniques in order to achieve its aims. However, it does propose a number of characteristics to be applied across the development lifecycle.

- **Use unambiguous notations.** Ambiguity makes it difficult to determine whether or not errors exist and misinterpretation is a source of error introduction. Using a notation that has a well defined and well understood semantics removes ambiguity. Such notations often benefit from tool support, which can assist in verification.

- **Take small steps.** By taking small semantic steps between stages of the lifecycle it is easy to demonstrate that one development stage has been correctly refined from its predecessor.

- **Use appropriate notations.** Accept that a given notation may be powerful at expressing certain system properties but clumsy for expressing others. The aim is to use notations that allow the system properties or behaviour to be expressed simply. Don't attempt to use a single notation if this results in key system properties being difficult to express. Awkward expressions can be difficult to interpret or verify. Similarly, use the most appropriate verification techniques at each stage. Expect the outputs at each stage in the lifecycle to be clear and simple to understand.

- **Don't repeat information.** Each stage of the lifecycle should have a well defined purpose and focus on the new detail being introduced rather than repeat information. It is then clear what information has been introduced and what needs to be verified — rather than wasting energy verifying that information has been correctly copied from one source to another. Duplication can also be expensive during maintenance as it may become inconsistent and thus a source of error and confusion.

- **Check each stage before progressing.** Each design step should be verified as soon as possible to eliminate errors introduced in that stage. Effective reviews are crucial; reviews should clearly identify what an artefact is being reviewed against and the purpose of the review. Where review checks can be automated — such as coding style checks — then tool support should be used early.

- **Justify decisions.** Document the justifications for why design decisions were made, why they are appropriate, and any arguments demonstrating correctness of the decision. Such justifications support future analysis — especially in the event of implementing changes to

a system, but more importantly the process of documenting what you do is highly effective at driving out errors during development.

- Solve difficult problems first. Manage development risks by solving difficult problems early. This also drives down the level of internal change that might otherwise be introduced if risks mature later.

Many of the approaches advocated here also contribute to the provision of strong verification evidence that, if collected appropriately, can contribute positively to the construction of a certification argument, demonstrating that the system has been built respecting safety or security needs. None of the concepts are new or radical; if anything it is the careful application of sound engineering practices using understood tools and techniques that has made this approach successful.

2.2 Using formal methods within the CbyC framework

The CbyC approach is particularly powerful when instantiated with formal methods and approaches. Formal methods have precise semantics and often have an associated language of reasoning that enables the user to unequivocally demonstrate the truth or otherwise of a property. Specification languages such as the Z Notation [Spi85] benefit from a richness of notation that allow the application to be described in terms of real world entities and relationships; Z supports both the concepts of refinement and encapsulation. In Z, data and operation refinement allow an abstract specification to be refined toward a concrete, executable realisation. Z's schema notation allows detail to be hidden except at the point of introduction and makes complex specifications manageable, giving focus to the aspects of interest at a given point in a specification and allowing the problem to be decomposed into small, manageable fragments. Notations such as CSP [Hoa85] are powerful for modelling and reasoning about concurrency problems, especially when used in conjunction with model checkers such as FDR [FDR]. SPARK is a subset of the Ada programming language enhanced with contracts that has a formal semantics and is supported by a suite of tools: the Examiner, Simplifier and Proof Checker, that allow conformance to language and program properties to be proven. All these notations (Z, CSP and SPARK) provide points in the development lifecycle prior to the production of object code, when there are artefacts with a clear semantics. This enables these artefacts, specification and design documents, or source code, to be formally verified, either as a refinement of a previous lifecycle phase, or more commonly, as possessing key properties.

Interestingly, many of the benefits of formal notations do not come from the application of verification techniques, tool supported or otherwise, but from the additional attention to detail imposed on the author when applying the techniques. Although tools can help to demonstrate (partial) completeness or correctness it is often before the point of application of such tools that benefits are first realised as the very act of expression within a formal notation causes the author to explore the problem domain with a logical mindset — thereby detecting and investigating incompleteness in the requirements early in the lifecycle.

Having said that, the ability to use tool support to automatically check properties of the system and even simulate aspects of the system under development is extremely powerful at detecting

early lifecycle errors and demonstrating properties of the final system to the customer or key stakeholders.

3 The Tokeneer ID Station Experiment

The aim of the Tokeneer ID Station (TIS) Experiment [TIS], commissioned by the US National Security Agency (NSA), was to determine whether it was possible to write software to the standards imposed by EAL5 of the Common Criteria [ISO99] in a cost effective manner.

The method by which the experiment was undertaken was for Altran Praxis to redevelop a well defined component of the existing Tokeneer System [RL98] using the CbyC approach applied using formal notations at every stage of the development lifecycle. Tokeneer was a system previously developed by the NSA as an unclassified demonstration of the use of smart cards and biometrics. CbyC was applied in the redevelopment of the core functions of one component of the Tokeneer system. The development was assessed against EAL5 of the Common Criteria to determine whether the approach achieved the necessary assurance evidence to certify a security system to EAL5. By monitoring the skills needed to perform each stage of the development approach and the effort involved it was also possible to establish whether the approach was cost effective.

The experiment was time boxed and some activities were not completed but an estimate of the cost to complete the activity was provided in all cases to allow the true cost of the approach to be determined.

3.1 The Tokeneer system

Tokeneer provides protection to secure information held on a network of workstations situated in a physically secure enclave. The Enrolment Station issues tokens to users. To do this it relies on a Certificate Authority (CA) to generate user ID Certificates and an Attribute Authority (AA) to generate attribute certificates containing clearance and privilege information and biometric information. The TIS provides protection to the enclave by checking whether the user is authorized to enter the enclave and adding a certificate to the user token that authorizes the user to operate on the workstations within the enclave. The workstations check the certificate added by the TIS station to determine whether the user is authorized to use the facilities it provides.

Once initialised, the TIS holds public keys for the CA and AA. The primary function of the TIS is controlling user entry. The entry process being as follows: the user presents a token to the TIS containing three certificates, the user ID certificate, a biometric certificate containing fingerprint data, and a privilege certificate containing the role and privileges held by the owner of the token; the TIS checks the validity of these certificates and ensures they are signed by known authorities. The user then presents their finger to a fingerprint reader and the TIS authenticates the user by comparing the biometric data on the token with a scan of the user's finger. If this data matches and the user privileges allow them access to the enclave then a further authentication certificate is added to the token, (this is a certificate of relatively short duration) and then unlocks the enclave door, permitting access. If at any point the TIS deems there to be a breach of security an alarm is raised. There are also a number of administrator functions that TIS offers to users

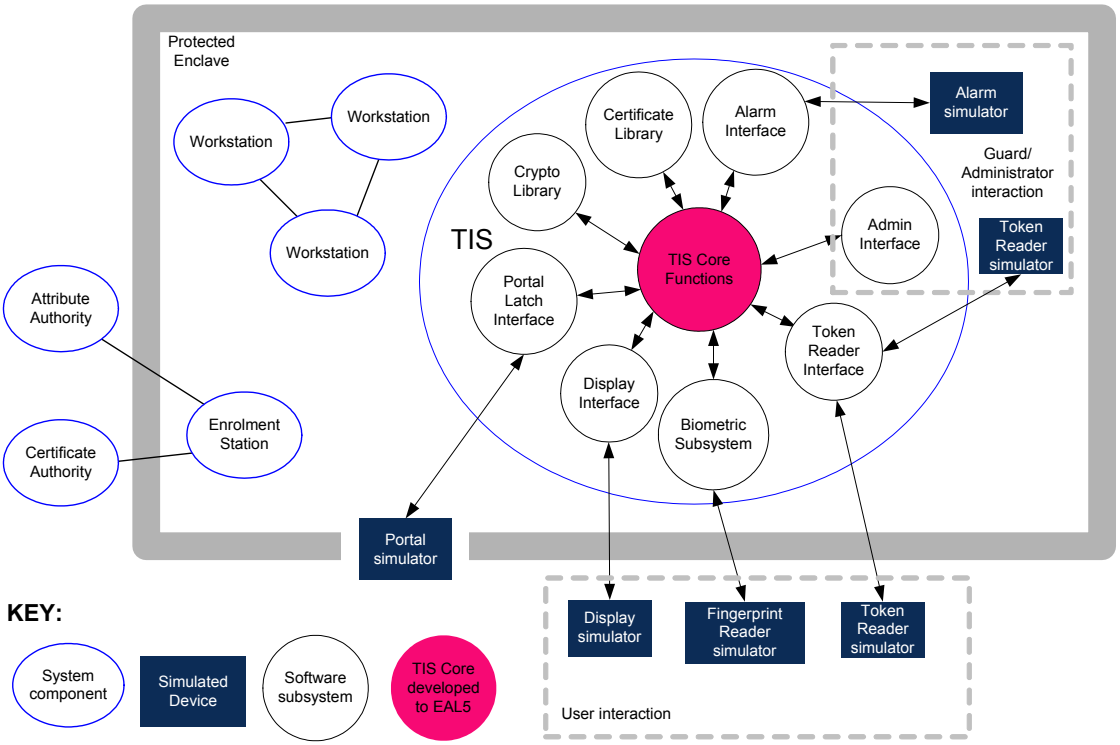

Figure 1: Overall Tokeneer System

with the appropriate roles. These are archiving log data of all transactions, overriding the door lock, and updating the configuration data which controls properties of the particular installation such as operating hours and security classification of the enclave.

Only the core functions of the TIS were developed using the full high integrity Correctness by Construction approach. Biometric and cryptographic components were simulated as were all external devices. The interfaces to external devices were developed using industry good practice but without the application of formal methods.

The customer introduced a change to the requirements part way through the design as a test of the robustness of the process. They added a requirement for the system to permit entry to the enclave to a user who had a valid authentication certificate on their token without needing to repeat the biometric checks.

3.2 The lifecycle

The TIS development lifecycle is depicted in Figure 2, it comprised six distinct phases: requirements analysis, security analysis, specification, design, implementation, and test.

Requirements analysis followed Altran Praxis' requirements engineering approach REVEAL [HRH01]. Key to this process was clear identification of the system boundary — important in this experiment was a clear understanding of boundaries between core functionality, to be developed to EAL5 criteria, supporting software, and functionality out of scope of the experiment

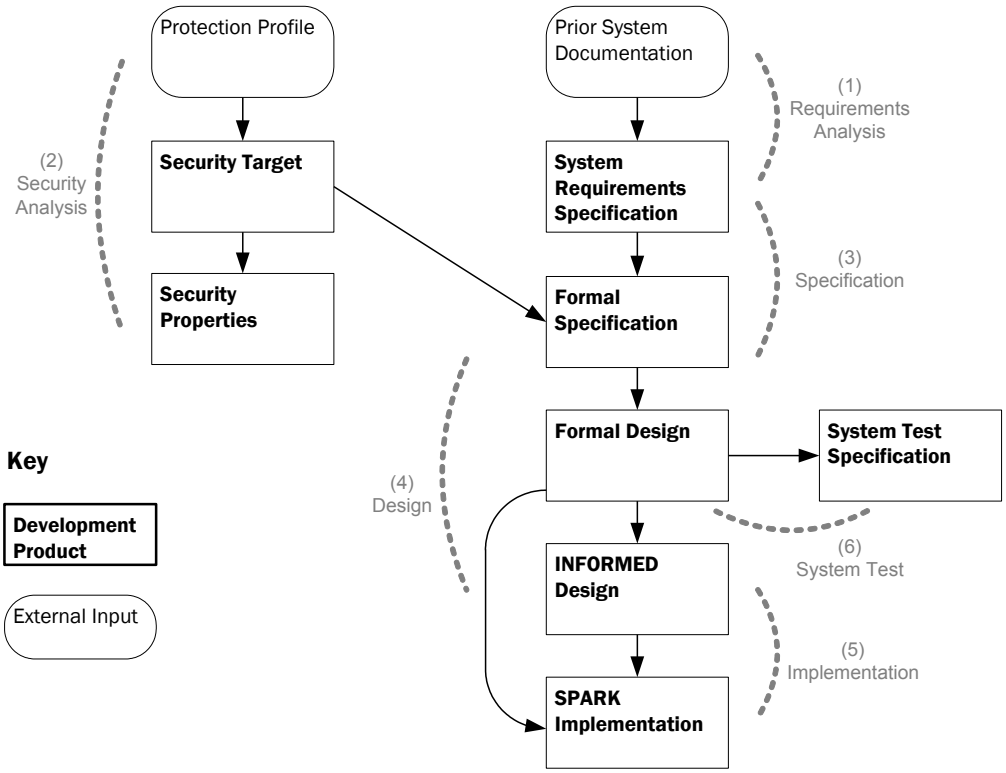

Figure 2: The development process

— for instance the original Tokeneer system additionally used a password in the authentication sequence. The context in which the TIS operated was also analysed giving a clear understanding of the TIS environment, such as the certificates generated externally and the way in which the door and its locking mechanism operated. Scenarios representing successful and erroneous interactions with TIS were developed with the customer to gain a clear understanding of the required behaviour of the system.

Security Analysis was performed orthogonally to the remainder of the development process, it responded to the supplied Protection Profile with a security target and development of the security properties required of the TIS. These activities focussed on the security needs of the system without consideration of the required user functionality. A key output of this activity was a Formal specification of the security properties developed using the Z notation.

Specification of the TIS took the form of a formal behavioural specification developed using the Z notation. The specification provides an abstract model of the system, focusing on interactions of the system with its real world interfaces, ignoring internal details. By developing a behavioural model of the system it was possible for the details of the proposed behaviour of the system to be presented early — before code production. With the help of customer review we were confident that we were planning to build the right system.

Design was divided into two components. The Formal design, again developed in Z, is a refinement of the specification introducing the internal details of how the system works — in

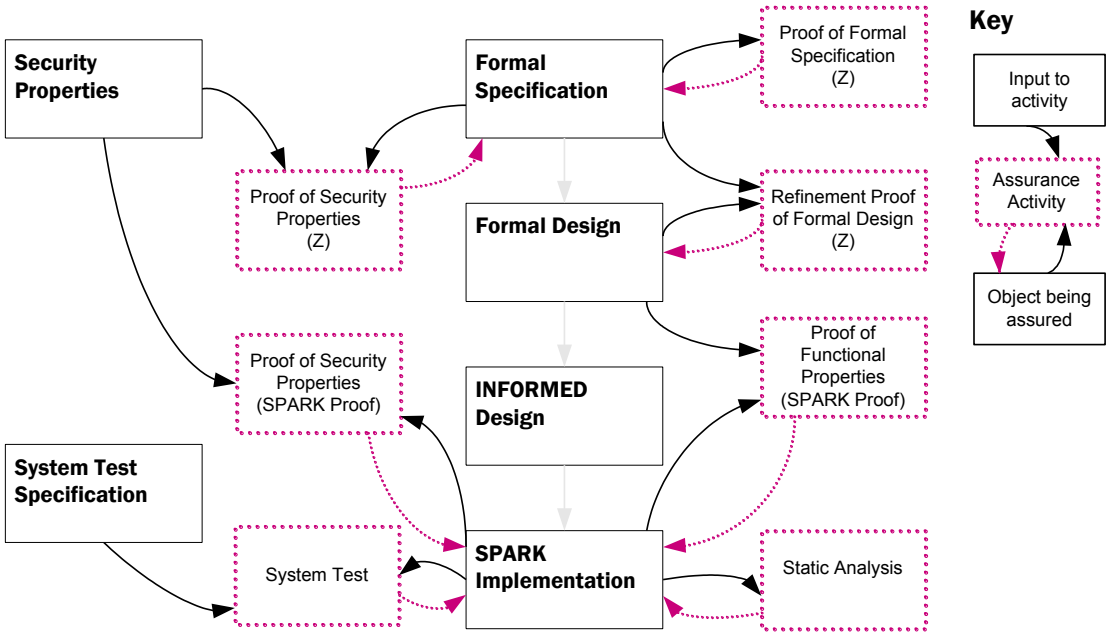

Figure 3: Assurance Activities

the case of TIS the design resolved some priority issues which led to the specification being potentially non-deterministic in its behaviour, additionally the details of logging and the structure of certificates as raw data streams were introduced.

The INFORMED design [Ame01] focused on developing a software architecture, it identifies implementation modules and the information flow between them, it apportions each component of the formal design to the program module that implements that component, it also covers file structures and constraints not covered formally.

Implementation of the core TIS is written in SPARK [Bar03] using both flow and proof contracts. Data and information flow analysis and proof of absence of run-time errors were done before code review and compilation. Implementation from the formal design was relatively straightforward — with simple mappings between predicates and code fragments.

Testing was limited to system testing, which was based on achieving a basic level of coverage of all the schemas in the Formal Design. Ordinarily this would have been undertaken with code coverage metrics being collected to ensure an adequate coverage of the source code had been achieved. The Formal Design provided a very clear definition of the required behaviour of the system on which to base tests.

The aspects of the implementation process that were more radical were the verification activities. These focused on verifying the correctness of each lifecycle phase early. Further, by using consistent Formal notations for the Security Properties, the Formal Specification and the Formal Design, it was possible to prove that the Formal Specification adhered to the Security Properties and that the Formal Design was a refinement of the Specification. The other area where proof was applied was in the code, in addition to proving the absence of run-time errors, some of the security properties were expressed as SPARK proof contracts, the code was then proven to con-

form to these properties. Figure 3 demonstrates the assurance activities undertaken, excluding review which occurs as each component is complete. Each assurance activity was undertaken as soon as all the inputs to the activity were complete and before proceeding to the next lifecycle activity allowing errors introduced at each phase to be driven out by more than just review scrutiny.

3.3 Results and subsequent scrutiny

The key outputs of the project were a 100 page behavioural Z specification; the core software comprised 9,939 lines of code with 6,036 lines of flow contracts and 1999 lines of proof contracts. The supporting software, written in Ada95, comprised 3,697 lines of code. The entire development required 260-man days, provided by three people working part time over 9 months. The productivity over the project as a whole was 38 lines of code per day, with the coding rate of the core software working out at 208 lines of code per day against a rate of 182 for the support software. Analysis [BC03] showed that the process had been developed to EAL5 and in some areas had exceeded the requirements of EAL5 particularly in the levels of formalism applied.

The whole project archive was donated to the Verified Software Repository in 2008 [TIS] and has subsequently been subjected to wide ranging scrutiny. To date, five defects have been found in the core software. These defects are fully documented in [WAC10] and were found through a combination of application of improved tools and critical review. Two of these are completely benign in the code as it stands, the other three represent potential insecurities in the software. Of these three, one would have been detected by the latest variants of the toolsets used on the project — assuming the most demanding levels of checks were selected, a further would have been detected by undertaking program proof of the remaining security properties and the last could have been detected following scrutiny of code coverage results.

These results are encouraging and suggest that, with the latest tools, the application of formal methods supports the development of high quality software suitable for critical domains. Of course, we can never be certain that every fault has been found but the level and variety of external scrutiny to date gives considerable confidence in the state of the Tokeneer core software.

Further, the results presented in [MW10] show that following extensive review of the whole code base and the use of CodePeer the most significant errors were found in the support software. This was written by the same engineers as the core software, but without the application of formal techniques such as SPARK and development from a formally specified design, giving a fair indication that the development process used on the core software did indeed produce higher quality software.

4 Challenges using formal methods in industry

It is clear from the results of Tokeneer that the application of formal methods can result in the efficient delivery of high quality software. However, the uptake of many of the approaches on an industrial scale has been limited. From a technical and commercial viewpoint this seems like a missed opportunity on the part of industry in general. To try and understand the reasons behind the apparent lack of industrial enthusiasm, the remainder of the paper seeks to establish

more abstract qualities of development and verification approaches which impact their successful adoption, taking as read that any formal approach will offer unambiguous notations and the opportunity for analysis of the system.

We propose that the following list is a representative, but not necessarily exhaustive, characterisation of desirable properties of any development notation, regardless of whether it constitutes a formal notation:

- scalable,
- notation approachable to all stakeholders,
- expressive (ease of capturing the problem),
- tool supported.

It is often the ability to satisfy these demands that influences the adoption of an approach, rather than the more obvious technical questions of whether the method or tools fulfil the goals of expressing the desired functionality and contributing towards a correct software implementation. In the following sections we consider these attributes in more detail and measure the success of the notations used in the development of Tokeneer against these criteria.

4.1 Scalable

This is a property that is well understood as being key to industrial applicability. There are two aspects to scalability, first whether the notation allows large problems to be expressed in a way that is still manageable to the authors and consumers of the artefact; secondly whether tooling associated with the notation is able to perform efficiently when processing representations of large problems. We look in more detail at the former problem. The problem of scalability is constant across the development lifecycle — a system that is complex is likely to have many requirements, a large design and a considerable code base. Effective notations offer encapsulation and modularisation which aid the presentation of information in manageable portions.

Tokeneer is small as industrial applications go. It has Altran Praxis' smallest Z specification covering full functional behaviour. Altran Praxis' most recent Z specification contains over 3000 schemas, the final developed system being of the order of 150KLOC of SPARK Ada demonstrating that the Z notation and SPARK are scalable. Larger SPARK developments have been undertaken outside of Altran Praxis.

In Z we can decompose the system state into logically cohesive components, developing structure within the system data model and allowing system behaviour to be decomposed into operations acting on a particular partition of the state. Overall system behaviour is achieved through composition of partial behaviours. This allows the participants of the specification to be able to contemplate the system using a divide and conquer approach, only ever needing to consider a small fragment in detail at any one time.

SPARK similarly allows the system to be analysed in fragments — making use of a rich package specification to allow components to be analyzed in isolation. Data abstraction also allows detail to be hidden from public contracts of a package and prevents contract proliferation.

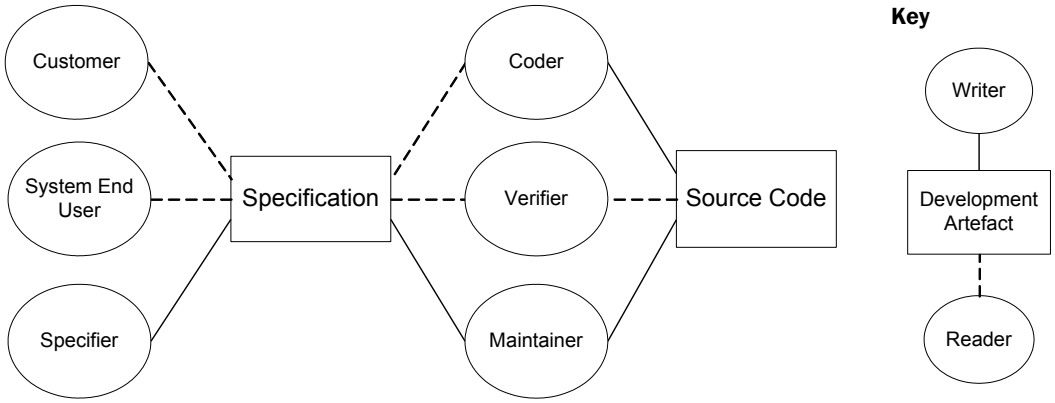

Figure 4: Artefacts and stakeholders

4.2 Approachable notation

A notation is considered approachable if it is usable by all those stakeholders who need to interact with it. The usability of a notation will depend on the familiarity of the notation — this familiarity can be acquired through use, although the ability to make such a transition to a notation will often be influenced by the underlying skills of the individual who needs to acquire the notation. To this end there are two things that influence the success of the notation to be approachable: the diversity of stakeholders who need to be involved with the notation and the difference between the notation and the languages already familiar to the stakeholders.

A system specification is likely to have a large number of stakeholders with diverse expertise, from end-users and customers to coders. The end-user and the customer are unlikely to be experts in the specification notation, although for the specification to be truly effective both the customer and the end-user will need to understand the system that is being specified — by doing so they will gain confidence that the system that is about to be built will offer the desired functionality. In the case of Tokeneer we were privileged to have a Z expert as our customer. However, where the customer and end-users are not experts in the notation we introduce a potential language barrier at a crucial early stage in the development lifecycle. It is at the point of developing the specification that we are first likely to uncover omissions from the requirements, details of corner case behaviours that the requirements don't define. Finding and resolving these at the point of specifying the system is highly efficient and reduces surprises in the system behaviour and increases the likelihood that we construct the desired product.

Tactics can be employed to reduce the language barrier — Altran Praxis has a policy of supplying a high level of English language description alongside the formal notation although reading just the (imprecise) English text will loose the value of the precise formal notation. Provision of training can be effective where there is not too great a disparity between customer, end-user skills and the selected notation, However, training requires a high level of customer commitment and can be problematic where the customer or end-user representation is large. Animation and scenario modelling are powerful as they allow demonstration of features of the system based on the specification, however a large specification can result in state space explosions and exploring all cases exhibited by the animation could be prohibitive in terms of time.

Even relatively simple aspects such as the documentation environment can prove significant hurdles in terms of familiarity of notation. For example the predominant text preparation method for Z is via the use of LaTeXwhile the industrial norm for document production is Microsoft Word or the like. In recent years tools have been developed to support the direct incorporation of Z paragraphs into Word documents [Hal08] thereby simplifying the process of generating documentation which incorporates textual descriptions, diagrams and formal paragraphs.

It is attractive from a commercial supplier perspective to obtain agreement to the specification and deliver to the specification; however this is only a practical proposition where the customer is truly engaged in the notation. A more realistic goal is for the specification to be viewed as an artefact internal to the development which allows pertinent questions to be asked of the customer or end-user; the questions being asked in a language familiar to the customer. Taking this approach we need to accept that it is highly possible that when producing a specification there will be differences of interpretation and that these differences may not be realised until the system is validated — this feels like a lost opportunity although it is no less powerful than using informal or semi-structured notations to deliver the system specification — where the notation would be insufficiently precise to detect many of the points of clarification that are uncovered when writing a formal specification.

Altran Praxis' experience with the use of Z as a specification language is that Z reading skills are easily acquired by coders and verifiers alike, suggesting that software engineers typically possess the necessary logical deductive skills appropriate to interpreting the Z notation.

By contrast the number of stakeholders involved with the source code, who might be required to understand notations associated with formal code analysis or proof are fewer. Furthermore, it is often possible to express the proof language in a semantics which represents a modest extension from the code semantics. There is a small semantic gap between the SPARK language and Ada making it a relatively painless transition for an Ada programmer to be able to correctly express and interpret SPARK contracts and the verification conditions generated by the associated tools.

4.3 Expressiveness

One of the fundamental characteristics of the CbyC approach to software development is to take small steps between lifecycle stages so that at no point is there a large semantic gap during the refinement from specification to code. Taking this idea back a stage further it is important to be able to describe the system in its real-world context as easily as possible in the specification. Often to achieve this we need to express complex properties of the system's interaction with the environment. To this end a highly expressive notation can be extremely effective, allowing a wide range of concepts to be captured without significant overhead of constructing building blocks that take the specifier's attention away form the problem domain and the task of expressing the behaviour of the system within that domain. Formal refinement techniques can then be used to transition from an abstract representation toward a design that can be simply implemented. However, the richer the language the harder it is to become an expert in the full language — this seems to be a true dilemma, not only to humans as users of the notation but to the provision of tool support to provide automatic verification.

Our experience in the development of industrial scale specifications is that the use of Z as a

highly expressive notation is extremely powerful in allowing the engineer to focus on capturing the correct description of the system's behaviour, without excessive distractions from having to find a way of encoding the relationships with a restrictive language.

Expressiveness becomes less of a critical characteristic of the notation as we move through the lifecycle toward code. Industrially used programming languages such as Ada and C and their language subsets such as SPARK Ada and MISRA C are sufficiently expressive to implement the system.

4.4 Tool support

One of the benefits of using formal notations is that they have sound semantics which make them amenable to tool supported verification, from the most basic syntax checking to automated or semi-automated proof. Without the underlying semantic definition it is difficult to make anything but basic checks on an artefact.

Automated verification is a highly powerful way of finding errors and inconsistencies in the outputs of the development lifecycle. Furthermore it is typically repeatable and should not be subject to human error. However, for automated verification to be cost effective, that is detect a sufficiently high density of faults in a sufficiently short period of time, there are a number of characteristics that need to be exhibited by the automated verification technique. The tools that support the technique need to be

- fast,
- trustworthy and supported,
- easily interpretable.

4.4.1 Speed

An effective verification tool must be sufficiently fast that the checks to be run repeatedly in a cycle of develop – check – correct – check. The speed of a tool is highly dependant on the modularity of the notation; the class of checks being undertaken and the amount of the system that the tool needs to interpret to enable it to perform its analysis. The speed of the Examiner is achieved by the analysis of one package body only being reliant on the enriched specifications of the other packages that are used by the package under analysis. The fuzz type checker [Spi] is fast due to the limited scope of its analysis. Both are sufficiently fast that they can be repeatedly run during development to ensure that the development output is being constructed correctly. Any checks that need to be run overnight cannot easily be used effectively as development is undertaken — although they can be used in the performance of final verification activities.

4.4.2 Correctness and Support

It is important to discuss correctness and support together as it is unlikely that any software product is completely fault free, but if support is readily available to handle faults found then the product can be considered fit for purpose. When a method and associated tools are selected for use on a project in industry the answers to the following questions will be fundamental to whether the tools are selected for use:

Can I get help in using the product?

Will the product be fixed promptly if I find a fault with it?

Will the product still be supported in 10 or 20 years time?

Will the product be considered appropriate by any certifying body?

If a development programme has chosen to include a tool in its development or verification strategy then training of personnel in the use of the tool and technology will be paramount, not knowing how to use a product to its best effect is expensive in time and a waste of the investment in the technology.

If a tool is found to be faulty in some respect then it is crucial for the development programme to either upgrade to a corrected version of the tool or fully understand the limitations otherwise there can be profound cost implications on the programme as a revised development or verification technique would need to be introduced. There is a widely held view that a product being open source means that it can be corrected, but this assumes that the source can be understood by the user. Even where the source for a tool is supplied there are significant costs and risks involved to anyone proposing modification to the tool.

Life expectancy of the tool suite is often of key concern to industrial developers. Many contracts include ongoing maintenance requirements and if the system is to be maintained then its development environment needs to be maintained and supported for the in service life of the software product. Although this is a risk with any tool, the risk is perceived to be greater where the tool is not itself available with a support contract.

Where the software under development is of a safety or security critical nature it is likely that a regulatory body will assess the processes, methods and tools used during development. Any tool where the output is used to gain verification credit will be expected to have an appropriate pedigree — either gained through a good history of use in the field, or by demonstration that the tool itself has been developed to a high standard.

4.4.3 Interpretation of output

Quality of the output of a verification tool dramatically impacts the time consumed analysing output and correcting inputs. Developments in tools to include hyperlinked renditions of the material analysed to aid navigation to the source of errors have been powerful at reducing analysis time. The Z Word tools [Hal08] do this to great effect allowing the user to run fuzz on the Word document and then jump from each error message to the source of the error in the Word document.

The level of false alerting of a tool can be crucial to its effectiveness, a tool that identifies a large number of potential problematic outcomes in the output will absorb a considerable amount of manpower in checking and justifying those cases that the tool could not provide a negative or affirmative outcome. One of the significant successes of the Examiner and Simplifier is the high percentage of verification conditions (VCs) generated through checking for absence of run-time errors that are automatically discharged. This makes the activity of checking the outstanding VCs manageable and has made the proof of absence of run-time errors in SPARK programs an option that is widely used.

5 Conclusion

Formal methods have a huge amount to offer industry in terms of providing unambiguous notations that are suited to formal verification that can in turn be automated. Many industrial standards for development of software at the highest integrity levels encourage the use of formal methods [ISO99, DEF97, EN 01] — to the point that it can be cheaper to conform to the standard by using a development approach that makes use of formal methods than relying on a test driven argument for certification. The results of the Tokeneer project are a clear demonstration that the application of formal methods is a cost effective route to the development of high integrity software. Despite this, the adoption of formal methods by industry is perceived as difficult. This paper has looked at some of the less technical aspects that influence decisions about the process by which software is developed and has considered why these aspects rather than the technical merits of the approach are likely to be significant barriers to acceptance of formal methods.

Of the four key industrial indicators for the acceptability of a general development notation considered in this paper, scalability and expressiveness are being addressed by formal methods. The approachability of the notation is more challenging where the notation becomes exposed to a wide range of stakeholders, so this indicator is most applicable to early lifecycle activities such as systems specification, where interaction with the customer or end user becomes necessary to establish the desired behaviour. A number of tactics have been explored that suggest that approachability of the notation can be addressed by careful choice of the manner of presentation.

This suggests that the most significant barrier to industrial acceptance is the availability of supported tools — there is a relative plethora of tools available open source that provide the desired levels of automation, however, this is insufficient. In an industrial context, the need for tool qualification, fitness for purpose arguments, training and ongoing support make the adoption of open source tools without support contracts too high a risk on exactly the classes of project that would most benefit from automated verification. To overcome this hurdle, formal methods tools need committed maintenance — this requires collaboration between industry and academia to place supported products in the marketplace at a price that allows adoption on both modest and large scale applications.

Acknowledgements: My gratitude goes to John Barnes, Rod Chapman and Neil White for their comments on the draft of this paper.

Bibliography

[Ame01] P. Amey. The INFORMED Design Method for SPARK. 2001. Available on request from Altran Praxis. http://www.altran-praxis.com/

[Ame06] P. Amey. Correctness by Construction. S.P8001.11.1. 2006. Available on request from Altran Praxis. http://www.altran-praxis.com/

[Bar03] J. G. P. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[BC03] J. E. Barnes, D. Cooper. EAL5 Demonstrator: Summary Report. S.P1229.81.1. Dec. 2003. in [TIS].

[DEF97] DEFSTAN 00-55 (Part 1). Requirements For Safety Related Software in Defence Equipment. Aug. 1997.

[EN 01] CENELEC BS EN 50128. Railway applications — Communications, signalling and processing systems — Software for railway control and protection systems. 2001.

[FDR] FDR2 refinement checker. Formal Systems (Europe) Ltd. http://www.fsel.com/

[Hal90] A. Hall. Seven Myths of Formal Methods. *IEEE Software* 7(5), 1990.

[Hal96] A. Hall. Using Formal Methods to Develop an ATC Information System. *IEEE Software* 13(2), 1996.

[Hal08] A. Hall. Integrating Z Into Large Projects: Tools and Techniques. In Börger et al. (eds.), *Short Papers of the ABZ 2008 Conference*. 2008.

[HC02] A. Hall, R. Chapman. Correctness by Construction: Developing a Commercial Secure System. *IEEE Software* 19(1), Jan. 2002.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HRH01] J. Hammond, R. Rawlings, A. Hall. Will it Work? In *RE'01, 5th IEEE International Symposium on Requirements Engineering*. 2001.

[ISO99] ISO 15408. Common Criteria for Information Technology Security Evaluation. 1999. Version 2.1.

[KHCP00] S. King, J. Hammond, R. Chapman, A. Pryor. Is Proof More Cost Effective than Testing? *IEEE Transactions on Software Engineering* 26(8), 2000.

[MW10] Y. Moy, A. Wallenburg. Tokeneer: Beyond Formal Program Verification. 2010. http://www.open-do.org/wp-content/uploads/2010/04/ERTS2010_final.pdf

[RL98] L. Reinert, S. Luther. TOKENEER User Authentication Techniques Using Public Key Certificates, Part 3: An Example Implementation. Technical report, NSA Central Security Service INFOSEC Engineering, 1998.

[Spi] J. M. Spivey. The fuzz type-checker for Z. http://Spivey.oriel.ox.ac.uk/mike/fuzz

[Spi85] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1985.

[TIS] Tokeneer ID Station EAL5 Demonstrator Project. http://www.altran-praxis.com/security.aspx

[WAC10] J. Woodcock, E. G. Aydal, R. Chapman. The Tokeneer Experiments. In Jones et al. (eds.), *Reflections on the work of C.A.R. Hoare*. Springer-Verlag, 2010.