# Correctness by Construction: Integrating Formality into a Commercial Development Process

Anthony Hall

Praxis Critical Systems Limited,
20 Manvers Street,
Bath BA1 1PX, U.K.
+44 1225 466991
anthony.hall@praxis-cs.co.uk

**Abstract.** This paper describes a successful project where we used formal methods as an integral part of the development process for a system intended to meet ITSEC E6 requirements. The system runs on commercially available hardware and uses common COTS software. We found that using formal methods in this way gave benefits in accuracy and testability of the software, reduced the number of errors in the delivered product and was a cost-effective way of developing high integrity software. Our experience contradicts the belief that formal methods are impractical, or that they should be treated as an overhead activity, outside the main stream of development. The paper explains how formal methods were used and what their benefits were. It shows how formality was integrated into the process. It discusses the use of different formal techniques appropriate for different aspects of the design and the integration of formal with non-formal methods.

## Background

Praxis Critical Systems Ltd developed the Certification Authority (CA) for the MULTOS [1] smart card scheme on behalf of Mondex International. The purpose of the CA is to provide the necessary information to initialise cards, and to sign the certificates that allow applications to be loaded and deleted from MULTOS cards.

The CA is a distributed multiprocessor system for which the main requirements were security and throughput. It was required to be developed to the E6 process and standards, since previous experience had shown that this process forced a rigorous development which helped clarify requirements and meant there were no late surprises in testing. Thus the process was important even though the product did not actually go through evaluation. To meet the development budget and timescale it was infeasible to build the system entirely from scratch and so use of commercial off-the shelf (COTS) hardware and infrastructure software was mandatory.

An overview of the development approach and some results from the project can be found in a more general paper [7]. The present paper concentrates on the use of formal methods within the project.

## The Development Approach

### Overview

The development was a single process in which the formal and semi-formal deliverables were fully integrated. Figure 1 shows the overall set of deliverables from the development process, grouped into the main process steps. Formal deliverables are shown with heavy borders.

### Requirements

We used REVEAL®, Praxis' well-tried requirements engineering method [6], to establish functional, security and performance requirements, including the threat analysis and informal security policy. In addition we developed a Formal Security Policy Model (FSPM) which is discussed in detail below.

### Specification and Architecture

Of the three deliverables from this phase, one, the formal top level specification (FTLS), used mathematical notation.

The user interface specification was a relatively formal document in that it defined the complete look and feel of the UI. We started by working with the system operators to prototype a user interface, which would be acceptable and reasonably secure in the rather difficult physical conditions of the CA. We then wrote a complete definition of the user interface using screen shots to define the appearance and state transition tables to define the behaviour.

The high level design covered several aspects of the system's structure including:

1. distribution across machines and processes;
2. database structure and protection mechanisms;
3. mechanisms to be used for transactions and inter-machine and inter-process communications.

Writing a formal high level design is not a well-understood topic, unlike writing a formal functional description. There are many different aspects to a design, and there are several problems in using formality:

1. No single notation covers all aspects. A formal design will at best need a mixture of notations.
2. Some aspects, such as distribution, do not have an appropriate formal notation at all.

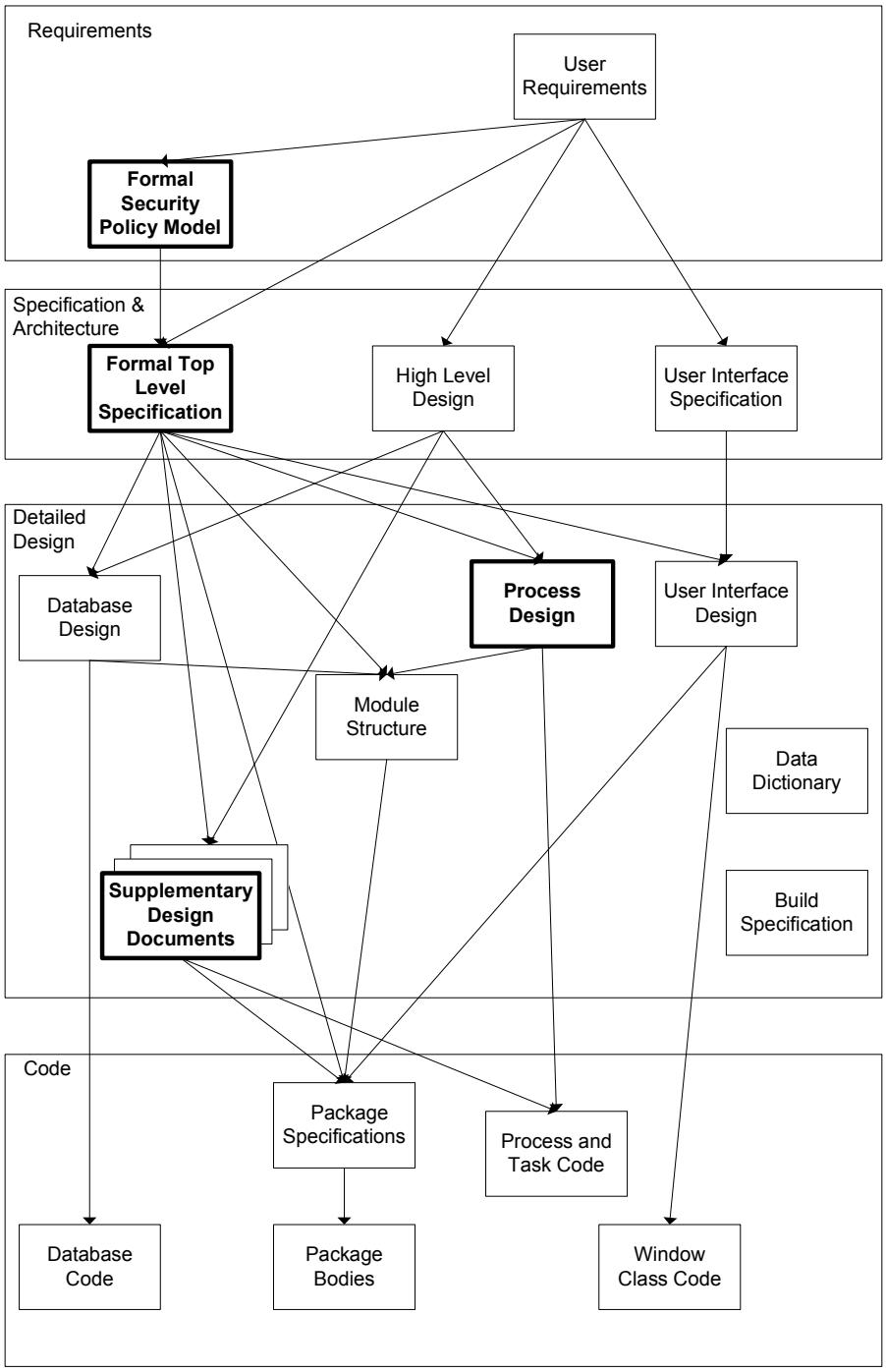


**Fig. 1.** Development Deliverables

3. Even where different aspects can be described formally, care is needed to relate the different formal notations.
4. The relationship between the formal design and the FTLS is not well understood: current techniques of refinement are not powerful enough for systems of this complexity.

The last point is particularly important. Conventional formal refinement in VDM or Z assumes that the structure of the design matches the structure of the specification, in the sense that it has the same set of operations. However, because of separation into processes, layering and other design techniques, the components of the design are structurally completely different from the components of the specification. Notations like B, which include programming constructs, can deal with some aspects of this restructuring but not, for example, with decomposition into concurrent processes. Overall we don't at present have any industrial-strength techniques for this sort of refinement in the large.

We therefore used semi-formal notations for the high level design. We then supplemented the semi-formal design by two formal techniques during the Detailed Design phase

**Detailed Design**

The detailed design expanded on the different aspects of the high level design. As with the HLD, most of the aspects were described in appropriate non-formal notations. For example the UI design described the window classes as simple state machines. We used formality for two kinds of deliverable:

1. Some of the modules had complex or critical behaviour. For example, the modules that managed low-level key storage were obviously highly critical and also had complex behaviour particularly at start up. We therefore specified the behaviour of these modules in Z.
2. We were very concerned about the concurrent behaviour of the system. Distributing the functionality across several processes on different machines introduces complexity and makes it hard to demonstrate correctness of the design. We therefore specified and analysed the concurrent behaviour formally.

**Code**

We used a mixture of languages for coding different parts of the system:

- The security-enforcing kernel of the system is implemented in SPARK Ada [2]—an annotated subset of Ada95 that is widely used in safety-critical systems, but has particular properties that make it suitable for the development of secure systems.
- The infrastructure of the system (e.g. RPC mechanisms, concurrency) is implemented in Ada95.
- The architecture of the system carefully avoids any security-related functionality in the GUI, so this was implemented in C++, using Microsoft's Foundation Classes.

- Some small parts, such as device drivers for cryptographic hardware, and one standard cryptographic algorithm, were used as-is.

We used automatic static analysis where possible, using tools such as the SPARK Examiner for Ada, and BoundsChecker and PC-Lint for C++.

From a formal methods point of view, the most significant choice was the use of SPARK for critical parts of the system. SPARK is one of the few languages that have a formal definition and thus meets the ITSEC requirement that the language "unambiguously define the meaning of all statements". Furthermore use of SPARK allowed us to prove automatically some key properties of the code, such as freedom from data flow errors.

For more discussion of SPARK and the choice of languages in this project, see [7].

### Testing

All our testing was top-down, as far as possible at the system level. Tests were systematically derived from the formal specifications and UI specifications. This systematic process coupled with the formality of the specification guaranteed a level of functional coverage. We measured code coverage and filled code coverage gaps with supplementary tests.

## The Formal Deliverables

### Formal Security Policy Model

The user requirements included an informal security policy, which identified assets, threats and countermeasures. We formalised a subset of the whole policy. Of the 45 items in the informal policy, 28 were technical as opposed to physical or procedural. Of these, 23 items related to the system viewed as a black box, whereas 5 were concerned with internal design and implementation details. We formalised these 23 items.

We used Z to express the formal model. We based our approach on CESG's Manual "F" [5] but simplified the method described there. The FSPM consisted of three parts:

1. A general model of a system.
   We model any system as a state plus a set of operations. This is a simplification of the Manual "F" model of a state plus transition relation.
2. A model of a CA system
   This is a specialisation of the general model of a system with some mapping between system state and real-world concepts such as users, sensitive data and so on.
3. A definition of a secure CA system .
   Each formalisable clause in the security policy is turned into a predicate that

constrains the CA system in some way. The overall definition of a secure CA system is one where all the constraints are satisfied.

There were four different kinds of clause in the informal security policy, and each kind gave rise to a different kind of predicate in the FSPM.

1. Two of the clauses constrained the overall state of the system. Each of these became a state invariant in the formal model.
2. Eight clauses required the CA to perform some function (for example, authentication). Each of these was expressed in the formal model as a predicate of the form:
   $\exists\ op : Operation \bullet property(op)$
   That is, there must exist an operation op with a particular property.
3. Sixteen clauses were constraints applicable to every operation in the system (for example, that they were only to be performed by authorised users). Each of these was expressed in the form:
   $\forall o : opExecutions \mid applicable(o) \bullet property(o)$
   That is, during any operation execution where the clause is applicable, the property must hold.
4. One clause was an information separation clause. We expressed this in the form:
   $\forall op : Operation \bullet start_1 \sim start_2 \Rightarrow end_1 \sim end_2$
   That is, for every operation if the starting states are indistinguishable from a particular point of view, then the final states should also be indistinguishable from that point of view.

The fact that only one out of the 25 clauses was about information separation is interesting because information separation is harder to express in Z than other properties, and that fact has led to suggestions that languages like CSP are more appropriate for expressing security properties. Indeed our formulation of the property is actually slightly stronger than is strictly necessary (we have essentially oversimplified the unwinding of all execution sequences). However it seems clear in this case at least that the benefit of using Z for the other 24 clauses, all of which are straightforward to express in Z, may outweigh its awkwardness in expressing information separation.

**The Formal Top Level Specification**

The FTLS is a complete description of the behaviour of the CA at a particular level of abstraction. It is complemented by a user interface specification that gives a more concrete description of the appearance of the system.

The FTLS is a fairly conventional Z specification. However, it contains some special features to allow checking against the FSPM. Whereas in conventional Z an operation is specified by one or perhaps two schemas, in the FTLS we used a number of different schemas to capture different security-relevant aspects of the operation.

1. We used separate schemas to define the inputs, displayed information and outputs of each operation. This allowed us to trace clearly to FSPM restrictions on what is displayed and how outputs are protected.

2. We used separate schemas to define when an operation was available and when it was valid. This is not directly a security issue but it allows us to distinguish those errors which are prevented by the user interface (for example by making certain options unavailable) and those which are checked once the operation has been confirmed and thus cause error messages to be displayed. In fact both kinds of condition must be checked in a secure way and since the user interface is insecure, some of its checks are repeated in the secure parts of the system.
3. We used an elaborate modelling of errors to satisfy the requirement that all errors be reported, whereas commonly Z specifications of error behaviour are underdetermined.

**Formal Module Specification**

We specified critical modules in conventional Z. Since these modules were internal to the system, issues such as what was displayed were not relevant, so we did not need the elaborate structure we had used for the FTLS.

The state specifications of these specifications were of course quite close to the actual machine state, and often the operation specifications used connectives such as sequential composition rather than the conceptually simpler logical connectives. We found, nevertheless, that the mathematical abstraction of this state, and the relative simplicity of Z operation specifications compared with implementations, was extremely useful in clarifying the exact behaviour of the modules.

**Process Design**

The CSP design was carried out at two levels: first we investigated the overall distribution across machines and major processes within machines. Then we checked the more detailed design of selected major processes.

For the top level design, we matched the CSP to the Z FTLS by mapping sets of Z operations into CSP actions. We also introduced actions to represent inter-process communications. This CSP model allowed us to carry out two kinds of check:

1. We checked that the overall system was deadlock-free.
2. We checked that an important security property was satisfied by the process design.

We checked that these properties were preserved even in the face of machine or software failure.

These checks were carried out automatically, by using the FDR tool from Formal Systems Europe. In order to use the tool we had to simplify the model by reducing the number of instances of identical processes, and we justified this reduction manually rather than by automatic checking.

We were able to find significant flaws in our first design by expressing it in CSP and checking its properties, and we have much greater confidence in the final design as a result.

At the second level we carried out detailed design of the processes on each machine. We then tried to prove that these were refinements of our high level design. Unfortunately it was not always possible to carry out these checks using the tool, because the processes were too big. However, we did increase our confidence in the lower level design by formalising it. Furthermore it was then extremely straightforward to implement the design using Ada 95 tasks and protected objects. We devised a set of rules for systematically translating CSP constructs into code. This was very successful and yielded code which worked first time, a rare experience among concurrent programmers.

## Relationships Between Deliverables

### Tracing

We used requirements tracing to record the relationships between different levels of requirements, specification and design. This gave us a syntactic link between, for example, a piece of Z specification and the requirement that it helps to meet and the parts of the design that implement it. Simple tracing links of this sort do not carry any semantic information, although it is possible to enrich them to give more information about the way in which later deliverables satisfy earlier ones [6].

### Relating Formal Deliverables

There are two kinds of relationship between the formal deliverables. There is a refinement relation between later deliverables and earlier ones: for example the FTLS should refine the FSPM. There is a more complex relationship between formal deliverables that cover different aspects of the system, for example between the FTLS and the process design.

We did not formalise these relationships. For example we did not carry out a refinement proof of the FTLS against the FSPM, although we did structure the FTLS to make informal checking of some aspects of the refinement straightforward.

Although we did not formalise the relationship between the FTLS and the process design, we did consider the nature of this relationship carefully.

It is sometimes suggested that in order to use different formal notations together, one must find a common semantic base that embraces both notations. However, the whole purpose of using different notations is to express different aspects of the system. One should expect, therefore, that there are aspects that are expressible in one notation but not in another. For example, Z allows us to express complex data structures while CSP allows us to talk about failures and divergences, which are inexpressible in Z (or, more strictly, in Z specifications using the established strategy).

On the other hand, clearly we need to find some link between the different notations; otherwise we are simply describing different systems. Our approach,

therefore, is to define some intersection between the notations rather than trying to find their union.

In this case we chose actions as the intersection between CSP and Z. Roughly speaking, a CSP action is the same as an operation in Z.

More precisely, the link between our process design and our Z specification involved two steps. First, we mapped sets of similar operations into CSP actions. Similarity, for this purpose, meant that the operations had the same security properties. For example there was a set of those actions that were both *security-critical* and *user-activated*: although these did very different things functionally, from a security point of view they all used the same processing path.

Having identified the significant sets of actions, we then broke them down to operations of the various processes in the system. These operations did not have Z specifications (since we did not do a complete Z specification of all modules below the FTLS) but they did correspond to sets of operations in the code. This second step is in fact a form of refinement.

### Relating Formal and Informal Deliverables

There was a very strong, but informal, relationship between the FTLS and the UIS. Every operation in the FTLS had a corresponding specification in the UIS. The UIS defined the lexical and syntactic levels of the operation, while the FTLS defined its semantics. Usually the UIS was finer-grained than the FTLS: for example an operation which appears atomic in the Z might involve several actions by the user to select values or type in text.

Most modules did not have formal specifications. In many cases, however, the module directly implemented a substantial part of the operation defined in the FTLS, so the tracing to the FTLS gave an "almost formal" definition of the module's behaviour. Where the module did have a low level Z specification, there was a direct correspondence between the operations of the module and those defined in the Z. We could have achieved even greater rigour by translating the Z into SPARK pre- and post-condition annotations. However, we did not feel that this gave us enough extra benefit to be worthwhile. The main reason was that the concrete state was usually more complex than the abstract state in the Z, but the relationship was very straightforward. The translation would therefore have been tedious but not very valuable.

The low-level CSP processes translated very directly into Ada tasks and calls to protected objects. This relationship, although again informal, is quite straightforward.

## Conclusions

It is clear that the use of formal methods in a development of this sort is practical. Furthermore it is our experience in this and other projects that well-considered use of formal methods is beneficial whether or not there is any regulatory requirement for it.

Specifically, our use of Z for the formal security policy model and for the formal functional specification helped produce an indisputable specification of functionality, which was the authority for all future development, testing and change control.

At the design level, we extended this formality by using Z, CSP and SPARK. This was also beneficial. However, not all aspects of design can be formalised and it is necessary to have clear relationships between formal and informal parts of the design.

The development has been successful. The number of faults in the system is low compared with systems developed using less formal approaches. The delivered system satisfies its users, performs well and is highly reliable in use. Productivity is within normal commercial range for systems of this type: formal methods do not add to the cost of the project. Furthermore, the benefits continue into the maintenance phase. The CA is now maintained by Mondex, and they endorse the advantages of a formal specification for maintenance, since it makes the specification of changes less ambiguous and greatly improves the chances of identifying side effects of a change on other parts of the system.

However, we must include two caveats. The first is that our use of formality did not extend to the use of proof, although in other contexts [3] we have used proof at both specification and code level successfully. The second is that the product has not in fact been evaluated against ITSEC E6, so that our view of whether we actually achieved E6 levels of assurance remains conjecture. Nevertheless, we are convinced that the rigour of E6 and in particular its emphasis on formality is beneficial for high integrity projects.

## Acknowledgements

The author would like to thank John Beric of Mondex International for his permission to publish this paper.

## References

1. See www.multos.com
2. Barnes, J., *High Integrity Ada - The SPARK Approach*. Addison Wesley, 1997. See also www.sparkada.com
3. Steve King, Jonathan Hammond, Rod Chapman and Andy Pryor *Is Proof More Cost-Effective Than Testing?*, IEEE Transactions on Software Engineering, Vol 26 No 8, pp675–686 (August 2000).
4. Information Technology Security Evaluation Criteria (ITSEC), Provisional Harmonised Criteria, Version 1.2, June 1991.
5. CESG Computer Security Manual 'F' – A Formal Development Method for High Assurance Systems, Communications Electronics Security Group, 1995.
6. Jonathan Hammond, Rosamund Rawlings and Anthony Hall, *Will it Work?* Proceedings of RE'01, 5th International Symposium on Requirements Engineering August 2001.
7. Anthony Hall and Roderick Chapman, *Correctness by Construction: Developing a Commercial Secure System*, IEEE Software, Jan/Feb 2002, pp18 – 25.