Tokeneer ID Station
EAL5 Demonstrator: Summary Report

S.P1229.81.1
Issue: 1.1
Status: Definitive
19th August 2008

Originator

David Cooper (Technical Authority)

Janet Barnes (Project Manager)

Approver

Martin Croxford (Line Manager)

Copies to:

NSA

Randolph Johnson

Praxis High Integrity Systems
Project File

Contents

Executive Summary

In order to demonstrate that developing highly secure systems to the level of rigour required by the higher assurance levels of the Common Criteria is possible, the NSA has asked Praxis High Integrity Systems to undertake a research project to develop part of an existing secure system (the Tokeneer System) in accordance with Praxis' own *Correctness by Construction* development process. This development work will then be used to show the security community that it is possible to develop secure systems rigorously in a cost effective manner.

Process

The development process applied to the TIS high-integrity development can be summarised in terms of the following key phases:

1 Requirements analysis (the REVEAL® process)
2 Formal specification (using the formal language Z)
3 Design (formal refinement of the Specification and the INFORMED process)
4 Implementation in SPARK Ada
5 Verification (using the SPARK Examiner toolset)
6 Top-down system testing.

At each stage in the process verification activities were undertaken to ensure that no errors had been introduced. These activities included review and semi-formal verification techniques applicable to the entities being developed.

Project Findings

The TIS development project has demonstrated that the Praxis Correctness by Construction development process is capable to producing a high quality, low defect system in a cost effective manner following a process that conforms to the Common Criteria EAL5 requirements.

The TIS system's key statistics are:

- lines of code: 9939
- total effort (days): 260
- productivity (lines of code per day, overall): 38
- productivity (lines of code per day, coding phase): 203
- defects (defects found post delivery per 1000 lines of code): currently zero, however independent testing is ongoing.

As well as achieving EAL5 levels of assurance, we believe that the Correctness by Construction process is close to achieving EAL7. The proof activity we use in our Correctness by Construction process is sufficient for EAL7, which involves tool supported code proof but manual proof of the Specification and Design. The process can be tightened appropriately to meet the additional quality control requirements of EAL7 by using tools that provide fully integrated electronic support.

To achieve the long-term aim of improving the take up of Common Criteria as a certification mechanism, the lessons learnt in this experimental development will have to alter the development practices of the majority of the NSA's contractors. This in turn will need the contractors to pass through four phases: understanding, belief, learning, experience. A continuing change management process will be needed.

1 Introduction

1.1 Background

In order to demonstrate that developing highly secure systems to the level of rigour required by the higher assurance levels of the Common Criteria is possible, the NSA has asked Praxis High Integrity Systems to undertake a research project to develop part of an existing secure system (the Tokeneer System) in accordance with Praxis' own *Correctness by Construction* development process, a high-integrity process developed by Praxis and applied by them on a number of commercial projects. This development work will then be used to show the security community that it is possible to develop secure systems rigorously in a cost effective manner.

Although the Common Criteria and its forerunners (the ITSEC scheme, the TCSEC — Orange Book, and others) have been in existence for a considerable time, there has been less use of them by industry than desired by their developers. Part of the reason for this may be that industry do not believe that it is possible to develop systems to the higher levels of certification in a cost-effective manner. Our experience at Praxis High Integrity Systems is that systems can be developed rigorously, and that this yields both a high-quality system, and lower cost.

1.1.1 Project Objectives

The key objective of this project was to obtain evidence of the applicability of the Praxis development process to EAL5-level system development. This includes two parts: feasibility (does it achieve reliable software?) and cost-effectiveness (is it cheaper than the traditional development process?).

Although this project has delivered a working system, the objective was not to have a new system *per se*, but to better understand the development process. The reason an actual system was developed was to give confidence that the development process does work in reality. It is also expected that this will help the NSA's desire to disseminate the results of this project widely through conferences, journals, and their own internal government communications media.

1.1.2 Statement of Work

The project objectives are laid down in the Statement of Work [10]. These can be summarised as follows (text repeated verbatim from the statement of work appears in italics):

The aims of this project were to undertake an experiment for *the introduction of formal methods (mathematically based) into the biometrics prototype called Tokeneer[9]. In addition this experiment hopes to demonstrate the practicality/cost effectiveness of employing the rigorous security standards for assurance as expressed in the Common Criteria.*

To attempt these goals Praxis High Integrity Systems in Bath, UK will redevelop part of the Tokeneer system- the Identification Station- with the SPARK Ada high integrity development processes. The subsystem includes functions of biometric authentication and smartcards in a networked system

requiring identification and authentication mechanisms. Requirements will be provided to Praxis in the document form known as a Protection Profile. In addition process and skills metrics will be collected from the Praxis process throughout the re-development in order to provide a point of comparison with previous (non-formal) development of this system.

The objectives of this project are to:

Analyze the operation modes and documentation of the Tokeneer prototype version 2 while concentrating on the Identification Station subsystem and its recently developed Protection Profile

Employ at minimum a Common Criteria compliant "semi-formal" approach to the modelling (design) and software development of a SPARK Ada based equivalent of the Identification Station. In addition this project should demonstrate not only the feasibility of building real projects following the Common Criteria Guidelines (at EAL 5) but also demonstrate the effectiveness of high assurance techniques like formal methods needed by Government systems to process classified information. The advantages of formal development will be assessed to include specifically a formal functional specification in Z of the core functions. Key security properties and formal statements about them should be realised. Another objective is to demonstrate how the formal functional specification possesses formal security properties and can act as a formal security policy model.

Thus, the design will correspond to the specification providing the formal statement of abstraction between the formal functional specification and the formal design specification. Z specifications of the functions will be constructed along with the definition of key security properties. The implementation will be demonstrated with SPARK Ada for the security functions, Interfaces with other components may be demonstrated through the use of stubs or simulators to pass and/or process I/O data.

1.2 Purpose

The purpose of this report is to report to the NSA comprehensive results from the development carried out in terms of the quality of the process used and the metrics collected, and to provide analysis of these results. There is the potential for using this report as a basis for further dissemination (e.g. conference papers).

1.3 Scope

This report gives a summary of the processes carried out and documents produced during the project. Full details of the project are in the project file.

As this project concentrated on demonstrating the Correctness by Construction development process, this report focuses on the development activities used for the TIS Core Functions. See section 2.2 for details of the split between Core Functions and Support Functions.

This application was produced to demonstrate the Correctness by Construction development process. However the development process does not address the issues of attaining the necessary operating

system security that would be required to ensure the application is protected from malicious attack. This was considered outside the scope of this project. If this application were to be used in a secure environment, operating system constraints would need to be applied to prevent user access to the files used by the TIS application. The installation does not address any of these issues and as such the application is known to be vulnerable to attack through modification of configuration data, keystore data and the audit log.

1.4 Structure

Section 2 explains the parts of TIS, and puts the Core Functions in perspective.

Section 3 describes the Praxis High Integrity Systems Correctness by Construction development process as applied to TIS.

Section 4 gives the metrics on effort, skills, errors, etc. recorded during the project.

Section 5 reviews the method in light of the conclusion that can be drawn from the metrics and in comparison to the requirements of the Common Criteria.

Section 6 looks at the next steps.

1.5 Glossary of Acronyms

AA	Attribute Authority
CA	Certification Authority
I&A	Identification and Authentication
LOC	Lines of Code
NIAP	National Information Assurance Partnership
NSA	National Security Agency
PP	Protection Profile
SPARK	Spade Ada Kernel [13, 14]
SPRE Inc	Software Process and Reliability Engineering, Inc (independent reliability testers)
SRS	Software Requirements Specification
ST	Security Target
TIS	Tokeneer Identification Station
VCs	Verification Conditions

2 Description of TIS / Tokeneer System

TIS is a single component of the larger Tokeneer System, and its context within this system is discussed below. All information in this section is summarised from the System Requirements Specification [1].

2.1 System Description

The system to which this project was applied was the Tokeneer ID station (TIS). TIS is one component of the larger Tokeneer system. The system as a whole provides protection to secure information held on a network of workstations situated in a physically secure enclave.

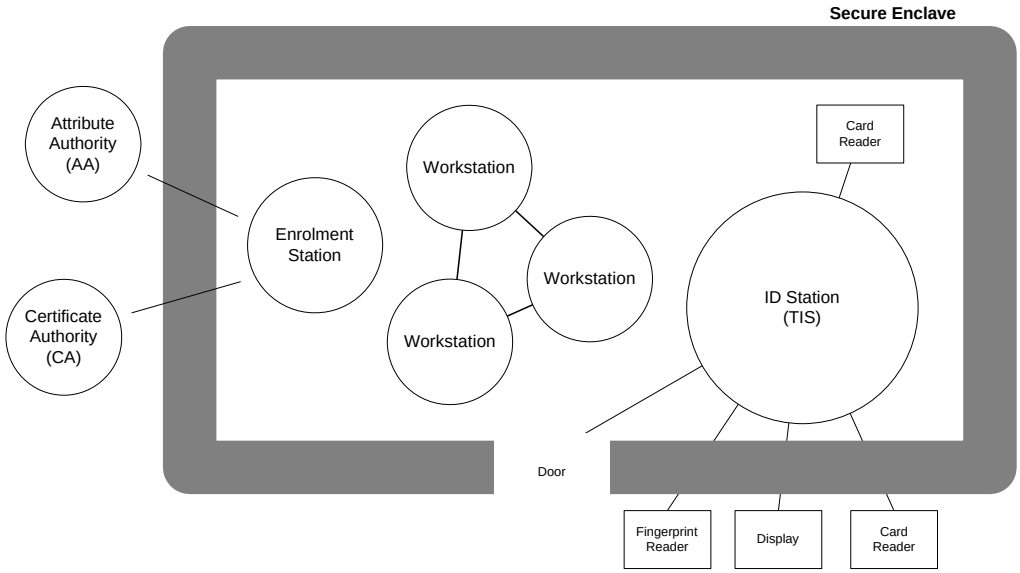

Figure 1: Overall Tokeneer System

The complete Tokeneer system consists of a secure enclave and a set of system components, some housed inside the enclave and some outside.

- An **Enrolment Station** is used to issue a **token** to an approved user. In order to generate the token the Enrolment Station relies on a **Certificate Authority** to generate a signed X.509 ID certificate and an **Attribute Authority** to generate signed X.509 Attribute certificates holding Privilege and Clearance information (Privilege Certificate) and Biometric information (I&A Certificate).

- The **Tokeneer ID Station** (TIS) is a stand-alone "trusted" entity responsible for performing biometric verification of the user. To perform this task it makes use of the biometric information in the I&A Certificate on the user's token and a fingerprint scan read from the user. If a successful identification is made then, assuming the user has sufficient clearance (held on the Privilege certificate), the TIS adds a signed Authorisation Certificate to the user's token and releases the lock on the enclave door to allow the user access to the enclave.

- The **Workstation** checks the Authorisation Certificate to determine whether the user is currently authorised to use the facilities it provides.

2.2 System Context

For the high integrity variant of TIS there are 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. These boundaries are expanded below:

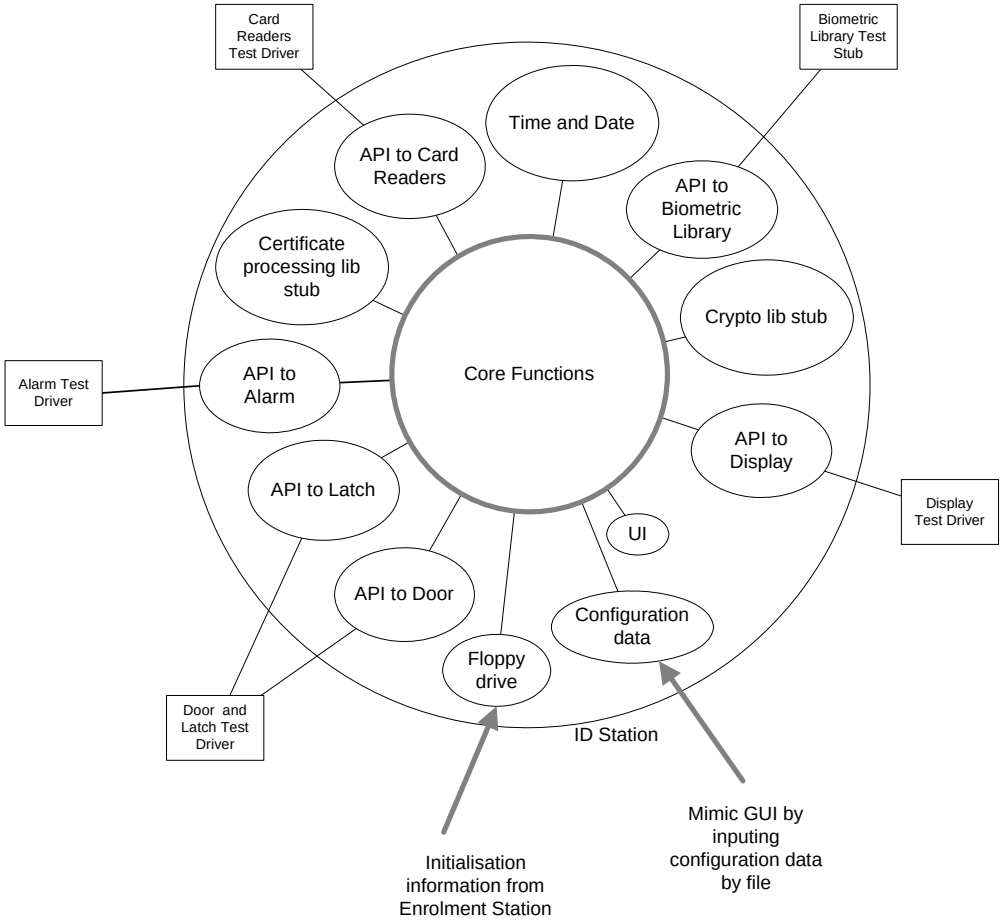

Figure 2: ID Station structure

The peripherals are mimicked by Test Drivers; the APIs provide message translation between the Test Drivers and the Core Functions; and the libraries are simple stubs. The GUI is mimicked by a file and simple command line interaction.

For the purposes of this development project, the developed software was divided into Core Functions and Support Functions. The Support Functions mimicked the drivers to peripherals by providing communications to external peripheral simulators, and in a real system would be replaced by bought-in drivers and peripherals, and hence would need to be evaluated to a suitable level of security. The

Biometric Library and Crypto Processing Library would also normally be bought-in products, but in this development were simulations to avoid licensing issues and to ease testing.

Praxis developed the Core Functions according to their Correctness by Construction development process. This is the part of the development that is being assessed for suitability against the high assurance Common Criteria requirements.

Praxis developed the modules within the ID Station but outside the Core Functions (except for Time and Date, which is operating system supplied) in a sound and professional manner, but not necessarily according to a high-integrity process. The APIs perform simple parsing of incoming messages and formatting of outgoing messages. The Library stubs mimic the behaviour of a Crypto Library (which stores keys and performs standard signing and verification operations) and a Certificate Processing Library (which extracts fields from certificates and constructs unsigned certificate contents from supplied data).

Peripheral Test Drivers (i.e. Card Reader Test Driver, Biometric Library Test Driver, Door and Latch Test Driver, Alarm Test Driver and Display Test Driver) were developed by SPRE Inc on a separate machine. Communication between the APIs and the Test drivers was via TCP/IP sockets. These test drivers were developed to model real peripherals as discussed below.

2.2.1 TIS Interaction with Interfaces

This section discusses the interactions made by TIS with the Test drivers and describes the interactions that these represent in the context of the Tokeneer System.

2.2.1.1 Card Readers Test Driver

This models two smart card readers, one located outside the Enclave (the User token reader), the other located inside the Enclave (the Admin token reader).

TIS reads the User's token from the User token reader and may write an additional Authorisation Certificate to the User's token via this smart card reader.

TIS reads an Administrator's token from the Admin token reader.

2.2.1.2 Alarm Test Driver

This models an audible alarm located within the enclave. This alarm is designed to notify a guard of a risk of a security breach.

TIS controls the state of the audible alarm (silent or alarming).

2.2.1.3 Door and Latch Test Driver

This models a sensor on the door into the Enclave indicating whether the door is open or closed, and the latch on the door, which can be set to either locked or unlocked.

TIS monitors the door sensor to determine whether the door is currently open or closed.

TIS controls the door latch setting it to either locked or unlocked.

2.2.1.4 Display Test Driver

This models the interface to a display outside the enclave, which provides information to a user wishing to gain entry to the enclave.

TIS controls the data presented to the user on this display.

2.2.1.5 Biometric Library Test Stub

This models both the Biometric Library and the interface to the Fingerprint reader, which would typically be performed via the Biometric Library.

TIS interrogates the Biometric Library to determine whether there is a fingerprint scan available for analysis and requests validation of the current fingerprint scan against supplied template information.

2.2.1.6 User Interface

A simple console and a facility to import configuration data from a file model the User Interface to the TIS console available within the Enclave.

TIS displays information on the console and reads simple keyboard input from the console.

Complex configuration updates, which might typically be performed via a GUI, are achieved by providing a facility to import configuration data from a file.

3 Approach

3.1 Method/Process applied to TIS development

The basic process applied to the TIS development is summarised in Figure 3. The key principle of this development process is to apply a philosophy of "Correctness by Construction". The crucial properties of this process are

1 being able to validate each lifecycle phase as early as possible

2 reducing the semantic gap between lifecycle phases so that the conformance of later lifecycle phases with earlier phases is provable.

These properties encourage the early detection and elimination of faults introduced during the development process.

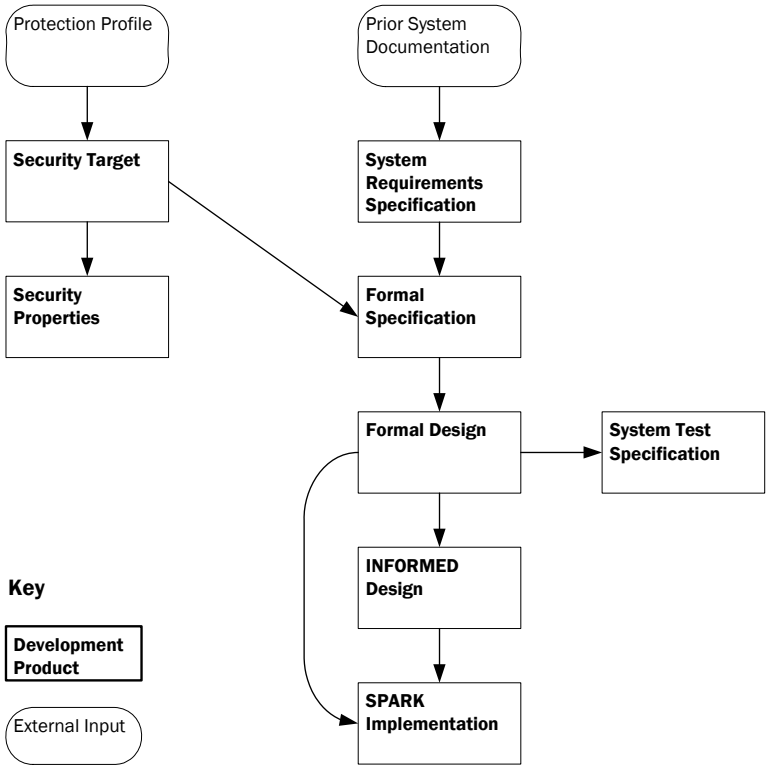

Figure 3 Development Process

Arrows show dependencies between activities to produce development products.

Full details of the activities performed within the development process are presented in the following sections:

Development Products	Section
Security Target	3.1.2
Security Properties	3.1.4
System Requirements Specification	3.1.1
Formal Specification	3.1.3
Formal Design	3.1.5
INFORMED Design	3.1.6
SPARK Implementation	3.1.7
System Test Specification	3.1.9

In addition to various review activities, there are a number of assurance activities performed to cross validate the various products of the development process. These are shown in Figure 4.

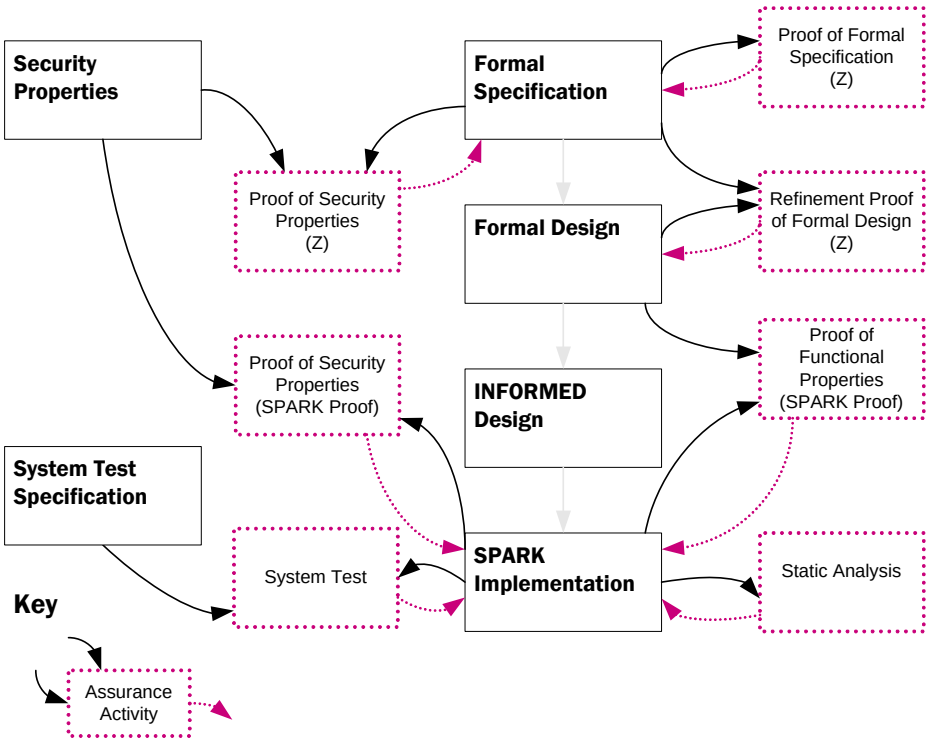

Figure 4 Assurance Process

Arrows into an assurance process indicate the inputs to the assurance process. Arrows out of an assurance process indicate the lifecycle product being validated by the assurance activity.

Full details of the assurance activities performed within the development process are presented in the following sections:

Assurance Activity	Section
Proof of Security Properties (Z)	3.1.4
Proof of Formal Specification	3.1.3
Refinement Proof of Formal Design	3.1.5
Static Analysis	3.1.7
Proof of Security Properties (SPARK Proof)	3.1.8
Proof of Functional Properties (SPARK Proof)	3.1.8
System Test	3.1.9

In addition to these formal assurance activities all outputs of the development process were reviewed as detailed in Section 3.1.10.

Finally the whole development process was supported by a Fault management process, which is used to manage the correction of faults in products of the development process. This is detailed in Section 3.1.11.

3.1.1 Requirements Analysis and Management

3.1.1.1 Approach

The aim of requirements analysis is to identify the needs of the stakeholders, the desired behaviour of the system, and any non-behavioural characteristics that are needed. Specifically capturing the security requirements in the Security Target is covered in section 3.1.2.The earlier these requirements are understood, the more likely the system will perform as the client expects.

Requirements management is a process that extends throughout the system development, but is most significant at the beginning, where it is used to identify

- the stakeholders, who have an interest in the development and use of the system
- the system boundary, to clarify the scope of the project and the interfaces to external systems
- the expected use, in terms of interactions between users and the system
- system properties, such as security properties, performance properties, etc.

As this project was “re-developing” a system that had already been developed to an extent by the NSA, the requirements activity was deliberately shortened and focused on clarifying only the differences expected between the existing system behaviours and the behaviours desired for the new system. The Praxis requirements process, REVEAL®, was used, although a full application of the REVEAL® process was not applicable because this was a re-development

A critical step in the requirements process for TIS was the identification of the *system boundary*. This was carried out on the first day of the project in a workshop between developers and client, and enabled us to make a clear separation between work being done on the core functionality (and hence would be developed to EAL5 criteria), work being done on supporting software (such as the simulators), and work outside the scope of the project. This also clarified the dependencies we had on aspects of the environment, such as certificates (supplied by the Certificate Authority) and the behaviour of the door/latch.

The behaviour of the system for the main interactions with people (user entry to the enclave, system administrator archiving the audit log, etc.) were documented using structured scenarios. These are easy to discuss with stakeholders, and rapidly help to bring everyone involved up to a common level of understanding of the system proposed.

The graphical user interface was agreed out of scope of this project. Normally requirements analysis would include the development of a user interface prototype, but of course this was not necessary for this project.

Requirements tracing was carried down throughout the project, with each level of system representation (requirements, specification, design, code, test) broken into uniquely identified trace units. Full tracing was carried out, in that every trace unit was traced back to the trace units in the higher level representation it implemented, but no tracing *analysis* was done, as the size of the project and the lack of significant requirements change did not warrant it.

3.1.1.2 Outputs

The output of this activity was:

- TIS System Requirements Specification [1]

3.1.1.3 Key Benefits

The reasons for producing the System Requirements Specification are:

- To clarify early in the project the system boundary (what is in scope and what is out of scope, and the interfaces necessary to external systems).
- To agree the requirements for the system with all of the stakeholders.
- To document the requirements in a sufficiently precise manner to allow subsequent development of the formal specification to proceed smoothly with little customer input.
- The clarify and document the assumptions about the behaviour of external systems (such as the door/latch, and smartcard), a common source of error.
- The identify and manage conflicting expectations between stakeholders.

3.1.2 Security Target

3.1.2.1 Approach

For security systems the Correctness by Construction development must be augmented with some steps directly concerned with understanding the security needs of the system and of achieving security certification. The Common Criteria requires a Security Target to define the security objective of the system and to justify the security design against these objectives. The Security Target may refer out to a Protection Profile: a reusable form of a security target. This was done on this project, referring out to an externally-supplied Protection Profile for a fully featured, fully implemented version of the TIS. The Security Target therefore concentrated on reducing the scope of the Protection Profile to the needs of this TIS development project.

3.1.2.2 Outputs

The outputs of this activity were:

- TIS Security Target [3]

3.1.2.3 Key Benefits

The reasons for producing the Security Target (and the Protection Profile on which it depends) are:

- It is a requirement of the Common Criteria.
- It identifies the key properties that must be shown to be upheld by the system for *security*. This separates out the concerns of functionality and the concerns of security, allowing more effort if desired to be given to the security aspects.
- It justifies the security measures to be implemented in terms of the threats the system is subject to or aiming to mitigate.
- It is the starting point for the Specification of the Security Properties (see section 3.1.4).

3.1.2.4 General Applicability

For the Security Target to be effective, it should focus the mind on the *key* properties the system must exhibit to be secure. The Threats/Assumptions/Objectives section of the security target or Protection Profile provides a justification for security measures in terms of potential security breaches the system is protecting against.

Due to the nature of this development project using simulated peripherals and reduced functionality the Security Target was very lightweight. In a normal EAL5 development considerable security analysis time would be given to the development of the Security Target.

3.1.3 Formal Specification

3.1.3.1 Approach

The aim of the Formal Specification is to describe unambiguously what the TIS system will do. It should enable the supplier and the client to gain a common understanding of what the system will do.

The choice of abstraction level is important. The formal specification should not address *how* the system is implemented, and in particular internal details are deliberately left very abstract. Interactions with the external environment are specified, but may be left abstract. For example, we provide an abstract model of the Audit Log (as this is exported as part of audit archiving), but no details of the structure, format or content of the log.

The Formal Specification was written in Z, a mathematical notation accompanied by an English narrative. 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 Z specification was checked using the *fuzz* type checker [11], a fast type checker that checks for consistency of types in all expressions.

The formal specification was developed by identifying *state* and *operations*.

All the state associated with the system was identified: this may be state held by the system (for example within TIS the configuration data is part of the system state) or state modelling the environment external to the system (for example TIS makes use of externally supplied time, and may modify the state of the door's latch). These two parts of the model are capturing different sorts of things. The state held by the system is a description of the *desired* state, whereas the modelling of the external environment is a description of the *actual* state of the world. This reflects the division identified during requirements capture between the requirements upon the system being built and expectations upon the environment. An example is the explicit constraint on the externally supplied time source: this was stated as always increasing. This explicitly identifies a dependency on the environment to supply a trusted source of time that never decreases — if time does decrease, the system is not required to respond soundly.

As the state was identified it was modularised, capturing tightly related state components within a schema, and adding invariants on the state as necessary. For example, in order to manage the relationship between the *door*, *latch* and *alarm* components, invariants were identified that described exactly the conditions under which the *latch* should be *locked* by the system and the *alarm raised* by the system.

Operations were then developed based on the scenarios identified during the requirements analysis. In the majority of cases these operations were identified as involving several phases. For example, the

UserEntry scenario involves receiving a token, validating it, possibly reading and validating biometric data, writing to the token, and finally unlocking the door. At each stage things may go wrong. Operation schemas were used to describe the successful behaviour first. Once the conditions for successful behaviour are identified, then the failure conditions can be deduced and the outcome of failure formally described. For example, biometric data can only be validated if it matches the template on the token (specified by the schema *ValidateFingerOK*). We then considered the failure conditions: either the Token is no-longer available to perform the match or the match fails (specified by schemas *ValidateFingerFail* and *UserTokenTorn* respectively). The overall process of validating the fingerprint data can then be presented as the combination of these three schemas.

In this way a full system description was generated.

Finally a viable set of values for the system state at initial startup was defined. This set of values must satisfy all invariants, and must represent a secure system.

In order to validate the Formal Specification a number of proof obligations were discharged:

- the existence of the initial state was proved
- explicit operation preconditions were identified, and proved to be the actual preconditions of the operations.

Precondition analysis characterises the conditions under which each of the operations can be performed. Each of these proof obligations was checked using rigorous argument (as apposed to formal, tool-supported proof). This provides a rigorous mechanism for checking the completeness and consistency of the specification. Although tool supported proof is possible for Z specifications it was not considered necessary to apply this level of formalism to TIS (and is certainly not required for EAL5 certification).

3.1.3.2 Outputs

The output of this activity was a single document:

- TIS Formal Specification [2]

3.1.3.3 Key Benefits

The reasons for producing a Formal Specification are:

- It provides an unambiguous description of what the system does. This is important for gaining client approval of the behaviour of the system to be developed.
- It is demonstrably complete.
- It is amenable to formal verification, i.e. it can be proved consistent.

The additional reasons for using the Z notation are:

- The notation is checkable, eliminating a number of minor errors in the production of the specification.
- The notation allows consideration of design detail to be postponed.
- The notation allows a large system to be de-composed into manageable sub-components.
- Tool support is available for type checking and proof.

3.1.3.4 General Applicability

The Formal Specification produced here was disproportionately large for the number of functions present in the system. The reasons for this were:

- We were mindful that the demonstration system had deliberately omitted many functions that would typically be present and we felt it would be instructive if the Formal Specification were structured to allow this omitted functionality to be added easily.
- Although the number of functions was small, the number of interfaces to the system was reasonably large and these were all modelled in the Formal Specification.

From past experience we would not expect the Specification of a larger system (offering more functionality) to be proportionally larger than the Formal Specification produced here.

3.1.4 Specification of Security Properties

3.1.4.1 Approach

The aim of this activity is to capture the system security properties unambiguously. These security properties are the key system properties that must hold of the system in order for it to satisfy its security obligations.

The security properties were expressed using the Z notation; the same notation as was used for the Formal Specification. The security properties were captured as proof obligations on the Formal Specification, so the same level of abstraction and context was used for expression of the security properties as was used in producing the Formal Specification.

By using the notation and context of the Formal Specification it was possible to prove that the Formal Specification exhibits the Security Properties. The proof took the form of an informal justification, with a discussion of the arguments required to perform each stage of the proof.

EAL5 does not demand proof of these properties, but a sample of the properties were proved to be held by the specification, and then later by the code. At the higher levels of certification such proofs are necessary, and can be carried out either rigorously by hand or using tool support.

3.1.4.2 Outputs

The outputs of this activity were:

- TIS Security Properties [4]

3.1.4.3 Key Benefits

The benefits of specifying the security properties using a formal notation are:

- It provides an unambiguous statement of the security properties.
- It ensures that we really do understand the properties we desire, and are agreed as such by the stakeholders.

The additional reasons for using the Z notation are:

- Assuming that these properties are expressed using the same notation and level of abstraction as the Formal Specification, it is possible to prove that the security properties hold of the Formal Specification.

3.1.4.4 General Applicability

It is not always possible to use the same model of the system for specifying the security properties as is used for the formal specification. This depends upon the security properties being expressed. For example, properties that must hold over a number of operations, or over arbitrary sequences of operations, will need a different style from that adopted here.

3.1.5 Formal Design

3.1.5.1 Approach

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 and carrying out some sample proofs can uncover the majority of errors.

3.1.5.2 Outputs

The output of this activity was a single document:

- TIS Formal Design [5]

3.1.5.3 Key Benefits

The reasons for producing a Formal Design are:

- It provides an unambiguous description of *how* the system does what the formal specification requires.

- It is demonstrably complete.

- It is amenable to formal verification, i.e. it can be proved consistent.

- By using the same notation as the Specification there is an opportunity for reuse where appropriate.
- Complex design decisions can be checked for correctness against the abstract behavioural description.
- By using the same notation as the Specification it is possible to prove that the design refines the specification.

The additional reasons for using the Z notation are:

- The notation is checkable, eliminating a number of minor errors in the production of the design.
- The notation allows implementable types to be modelled.
- The notation allows a large system to be de-composed into manageable sub-components.
- Tool support is available for type checking and proof.

3.1.5.4 General Applicability

Ideally, the formal specification should be a fully detailed description of the black-box behaviour of the system, and the formal design should explain how this behaviour is implemented internally. Such a division of responsibilities works particularly well for larger systems in which there are difficult design decisions being made that are invisible beyond the system boundary. The desire is to have a clear separation between externally visible behaviour and internal design.

This project did not demonstrate this separation very well. The division was blurred, because some externally visible behaviour was not elaborated until the design document, such as the details of the audit records and the priority of the operations.

3.1.6 INFORMED Design

3.1.6.1 Approach

The Aim of the INFORMED Design is to provide architectural and other non-functional information required to progress from the Formal Design to the Implementation in SPARK. To understand the importance of INFORMED you must first understand the properties of SPARK [13, 14].

SPARK is a programming language based on a sub-set of the Ada language, SPARK exploits the strengths of Ada while eliminating all its potential ambiguities and insecurities. A SPARK program has a precise meaning, which is unaffected by the choice of Ada compiler and can never be erroneous. These desirable goals are achieved partly by omitting some of Ada's more problematic features (such as unrestricted tasking) and partly by introducing annotations to capture the code designer's intentions. The combination of these approaches allows SPARK to meet its design objectives, which are: rigorous

definition, simple semantics, security, expressive power, verifiability and bounded resource requirements.

When used throughout the development process, SPARK can also have a beneficial effect on designs. Consideration of information flows at the design stage leads to programs with the desirable properties of abstraction, encapsulation, high cohesion and loose coupling. The complementary INFORMED design method exploits SPARK's properties to meet these goals

The target language, SPARK, is strictly hierarchical and highly modular, and a successful SPARK implementation is highly dependant on the localisation of state, primarily because the SPARK language makes the state interactions visible in special annotations. Good localisation of state results in meaningful annotations. The INFORMED Design stage provided a process for constructing a software architecture that focused on the location of the state within the system. This process helped the developers to understand the transformations necessary in passing from Z schemas to Ada Packages. It also determined which packages contain global state and allocated subprograms to packages. At this stage it was important to relate the state and subprograms that will appear in the implementation to the state and operation schemas that appear in the Formal Design.

The INFORMED Design also addressed design issues for which a formal treatment is not appropriate. For instance, in the case of TIS we gave details of the file formats within the INFORMED design as there was no value to be gained in presenting these formally. We also expanded the treatment of System Faults, which result from failures of peripherals — in the TIS Formal Design we chose not to model the potential failure of each peripheral although we did allow for system faults to be audited. Finally, in the INFORMED Design we imposed upper bounds on values in the Formal Design that were not bounded; in the Formal Design many state components were represented by unbounded integers (such as the count of failed entries), which needed to be bounded in the implementation.

Once the overall system architecture had been outlined in the INFORMED design we could produce SPARK specifications of all the packages in the system.

The INFORMED design served two purposes; first it provided an architectural framework in which to perform the implementation, secondly it aids maintenance and upgrades of the software by providing a route-map from the Formal Design to the code.

3.1.6.2 Outputs

The output of this activity was a single document:

- TIS INFORMED Design [6]

3.1.6.3 Key Benefits

The reasons for producing an INFORMED Design are.

- It focuses on the system architecture and ensures that the architecture fits the SPARK model.

- It provides the mapping from the Formal Design to the Code prior to writing the code.
- It complements the Formal Design without duplicating functional information.

3.1.7 Implementation

3.1.7.1 Approach

Implementation in SPARK started with producing package specifications based on the architectural information in the INFORMED Design and the functional information in the Formal Design.

A SPARK specification contains the Ada signature for all public operations and *annotations*, which specify the abstract global state held within the package, the global state used or modified by the operation and the relationship between the global state and the parameters, known as the *derives* information flow relationship. The derives annotations specify which imported (used) parameters and global state components may influence the final values of each exported (modified) parameter or global state component. These derives annotations are written based on the Formal Design. As the code is written they provide a basic check of conformance between the code and the expected flow relations.

Static Analysis of the package specifications ensured that the designed architecture satisfies the SPARK language constraints.

Once package specifications were written package bodies were developed. These contain the implementations of operations and the concrete declarations of package state. As each operation was written it was analysed using the SPARK Examiner: this checked the data and information flow properties of the code against the annotations provided in the specification and provided a way of checking the code early (before it can be compiled). This is a very effective way of eliminating a number of classes of errors such as the failure to set exported state on all paths through the code or using un-initialised variables.

Subprogram implementation was performed directly from the Formal Design. The Formal Design is sufficiently detailed to ensure that the mapping from the design to the code is simple; this is demonstrated in the Code Verification Summary [7].

Once the code for a package was complete the SPARK Examiner was run to check for run-time errors. This allowed the code to be demonstrated free from errors that might cause a run-time exception to be raised; such faults include accessing arrays outside their bounds or the overflow of numeric types.

The code for the package was then compiled. Developer testing was carried out on the compiled code if necessary.

The order of development of the system was a major consideration in our approach. Packages providing infrastructure were developed early and the development order was selected to introduce system level operations in an incremental manner. This means that a basic system can be built as soon as possible and functionality is added in subsequent builds. This has the advantage of addressing code integration risks as early as possible. For TIS this meant that the User Entry operation and all peripherals to support

this operation were implemented before the administrative operations and their supporting infrastructure.

3.1.7.2 Outputs

The outputs of this activity were:

- Ada Source code files.
- Proof Justification Files for proof of absence of run-time errors.
- Executable for TIS, **tis.exe**.

3.1.7.3 Key Benefits

The reasons for using SPARK as the implementation language are:

- SPARK is the only implementation language that truly meets the Common Criteria requirement to have an unambiguously, formally defined language (ALC_TAT.3).
- Code can be statically analysed very early — well before it can be compiled.
- SPARK is strongly typed giving added compile-time checks on the code.
- SPARK is a modular language allowing abstraction of detail.
- SPARK makes the flow properties of the code (at an abstract level) visible and checkable.
- SPARK can be demonstrated free from a wide range of errors without even running the code.
- There is tool support for comprehensive range of static analysis using the SPARK Examiner.

3.1.8 Code Proof

3.1.8.1 Approach

Two types of code proof can be carried out: functional and security properties. Functional code proof demonstrates that the code accurately implements the functions defined in the Formal Design. Security Properties proof shows that the code possesses the abstract properties identified in the Security Properties document, and expressed as theorems about the formal specification.

The code proof activity focussed on proving some of the key security properties of the code. Functional proof of the code was not performed for the following reasons:

- The step from Formal Design to code turned out to be very small, so it was very easy to check behavioural correctness of the code by review of the code against the Formal Design.

- Budgetary limitations on the project only permitted a sample of the proof activity to be undertaken.

It was therefore considered most worthwhile to prove security properties. Preservation of the security properties is less obvious from reading the code so failure to preserve security properties would be correspondingly difficult to demonstrate through code review.

In order to prove that a security property was preserved by the code the formal statement of the security property as stated in the specification of the Security Properties [4] was reformulated using the SPARK predicate language in terms of **pre** and **post** conditions. These security properties were inserted into the code annotations expressing proof contexts that had to be satisfied by the TIS core program. As the security properties have to hold of the whole system, these proof contexts are placed in the TIS main program and proved of the system as a whole by applying a divide-and-conquer approach to determine the necessary proof obligations that must be held by the subprograms used to implement the whole.

By using the SPARK Toolset it was proven that the code did implement the security properties. This activity involved using the SPARK Examiner to generate all the VCs (Verification Conditions) that need to be satisfied in order to prove that the code satisfies the properties stipulated within the proof contexts. These VCs were then passed through the SPADE Simplifier, which reduced the majority of the VCs to *true*. Outstanding VCs were then validated, either by review or by use of the Interactive Proof Checker Tool.

A full summary of the code verification activity, including code proof, is presented in [7].

Praxis have used full functional correctness proofs on other projects, and this has been found to be effective in finding errors. Depending upon the system in question, proof can be more cost-effective than unit testing in identifying errors, although it does not remove the need to carry out system tests, see for example [15].

3.1.8.2 Outputs

The outputs of this activity were:

- SPARK proof contexts inserted into the source code (see 3.1.7.2).
- Proof justification files for VCs that were not discharged using tool support.
- Proof scripts and rules for the Interactive Proof Checker for VCs that could not be discharged automatically.
- TIS Code Verification Summary [7]

3.1.8.3 Key Benefits

The reasons for performing formal verification are:

- Functional behaviour and system properties can be expressed formally independently of the code (e.g. in the formal design or in annotations), and then the code can be proved to conform to the specified behaviour.
- Proof is not limited to specific test cases, but demonstrates correctness across all possible inputs.
- Proof does not need to be a post-hoc activity, it can often be applied to partially developed systems to ensure functional correctness of aspects of an implementation that may be considered otherwise difficult to map to the design.

The additional reasons for using the SPARK language and proof toolset are:

- The proof language is easy to understand, being an enhanced dialect of Ada.
- The properties specified in SPARK proof annotations can be tailored to the required level of detail.
- SPARK tool support makes the majority of the proof effort automatic.
- SPARK proof can be applied to individual operations and thus is applicable before a system is complete, or even compliable.

3.1.8.4 General Applicability

In general, a project will have to decide what level of code proof should be carried out. On this project we chose to prove some of the security properties all the way down to the code primarily as a demonstration exercise.

If properties have been well captured at the same level of abstraction as the formal specification, if refinement from one level to the next is proved, and if the property described is preserved by refinement, then the benefit of proving the properties themselves at each level of abstraction is reduced. If, however, a property is not preserved in general by refinement (such as an information flow property) then direct proof at code level is powerful.

3.1.9 System Test

3.1.9.1 Approach

The aims of System Test are to demonstrate that the system has the correct behaviour as specified in the Formal Specification. This differs slightly from the goals of acceptance testing which is designed to demonstrate that the System meets its requirements. System Testing aims to achieve 100% coverage of the formal specification, so all possible behaviours described in the formal specification should be exercised at least once.

On this project we chose to perform system testing against the Formal Design rather than the Formal Specification. This was because there were aspects of the system that we wanted to test, such as the

values written to the audit log, which where not elaborated in the Formal Specification but were elaborated in the Formal Design.

The System Tests took the form of scenarios that might occur in typical usage, such as “Enrolling TIS” or “Administrator enters enclave and gains an Authorisation Certificate”. Both successful and unsuccessful operations were considered in order to cover all success and failure cases presented within the Z schemas in the Formal Design.

All System tests were documented in a System Test Specification prior to their execution. This included documentation of the expected outcome of the test in terms of visual indications, changes to stored data and new audit log entries. System test documentation also traced each test to the components of the Formal Design that the test attempts to exercise.

During review of the System Test Specification analysis of the tracing was performed to ensure that full coverage of the Formal Design was achieved.

System test execution was aided by the production of a simple program capable of executing test scripts. The test scripts prompted the tester when it was necessary to interact with the TIS interface and updated the test environment, for example by simulating the insertion of a token via the test drivers interfacing to TIS. By scripting the tests it made the tests repeatable and reduced the risk of human error during test execution.

Where code coverage metrics need to be captured, this would be done during System test. This allows us to question the use of any code that cannot be covered by a system test. Such code could be covered by adding focused unit tests where required. We did not capture code coverage during this project.

3.1.9.2 Outputs

The outputs of this activity were:

- System Test Specification [8]
- Test scripts for execution of each of the system tests.
- Test harness executable **testtis.exe** for running the test scripts.

3.1.9.3 Key Benefits

- System test focuses on testing the behaviour of the whole system against the expected (specified) behaviour.
- System test is likely to pick up faults due to erroneous interaction between modules within the implementation as it tests the system as a whole.
- System testing complements static analysis, in that it confirms the dynamic behaviour.

- A combination of static analysis and system test can make general unit test ineffective at finding faults, in that most faults that could be uncovered by unit test are also detected using static analysis or system test, resulting in general unit test not being cost effective.

3.1.9.4 General Applicability

The amount of time spent on system testing in this project was significantly less than would normally be expected. The reasons for the low testing effort are identified as follows:

- We did not measure source code coverage during system testing; this activity usually results in additional system (or unit) tests being added to the test suite.

- Running system tests usually requires the production of a test environment, often separate from the normal operating environment or as a harness enclosing the normal operating environment. Due to the nature of the Test Devices developed by SPRE there was no need for Praxis to perform this activity. The effort expended by SPRE developing the drivers would normally be included in the cost of testing.

- The environment in which the Core TIS functions reside is unusually constrained. For example the certificate processing library (which is outside of the Core) does handle a wide variety of faulty certificates and to test this library it would be necessary to use a large number of faulty certificates in tests; but the core itself either receives a well structured certificate or a fault indication and this is independent of the certificate field that is faulty, reducing to two the number of test cases required to fully cover the core's interpretation of a supplied certificate.

Correctness by Construction would usually test against a fully detailed system specification, using test coverage tools to ensure 100% coverage, and fully-automated testing to ease regression testing.

Although as discussed in section 3.1.7 static analysis and code proof is usually more effective at finding errors than unit testing, the full application of system testing is still necessary to uncover the faults associated with integration.

We have used code coverage tools on other projects at Praxis, and we have used information from these tools to guide the choice of unit testing. In general, we reserve testing for identifying errors of integration, and use static analysis and proof for identifying errors of behaviour in individual modules. Only when our incremental integration approach fails to allow all behaviours to be tested adequately at a systems level do we use targeted unit testing.

We used partially-automated testing. Normally, Praxis projects use fully-automated tests, which encourages frequent regression testing as the incremental integration proceeds.

3.1.10 Review Process

3.1.10.1 Approach

The following technical reviews were undertaken during the project.

The reviews were always undertaken by an independent member of the project team, (ie not the person who authored the material under review). The purpose of each of these reviews is tabulated below. In addition review feedback was sought from the Client for all deliverable documents, ie System Requirements Specification, Security Target, Formal Specification, Formal Design and INFORMED Design.

Review teams were always small. This reflects standard practice in Praxis, as it is important that the review is performed by people who have appropriate technical understanding and expertise. Where one person cannot bring all the necessary skills to the review then the review team is widened to ensure that all necessary skills are represented in the review team.

Review of	**Against**	**Purpose**
System Requirements Specification	Stakeholders' knowledge.	To confirm the system requirements with stakeholders.
Security Target	Protection Profile	To ensure that all security issues captured in the Protection Profile are covered.
Formal Specification	System Requirements Specification	To ensure that specified system satisfies the system requirements and that all tracing to the System Requirements is appropriate.
	Security Target	To ensure that all security issues that require functional consideration are addressed.
Proof of Formal Specification	Formal Specification	To check that proof obligations are appropriate and the proof argument is sound.
Security Properties	Security Target	To ensure that the security properties capture all security issues that can be expressed as a system property.
	Formal Specification	To ensure that the security properties are captured in the same context frame and level of abstraction as the system is formally specified.
Proof of Security Properties	Formal Specification Security Properties	To check that the proof argument is sound.
Formal Design	Formal Specification	To ensure that the formal system design is a refinement of the system specification and that all tracing to the Formal Specification is appropriate.
INFORMED Design	Formal Design	To ensure that all state and operations required by the Formal Design has been captured in the system architecture.

Review of	Against	Purpose
Code *including Static Analysis results.*	Formal Design	To ensure that the code implements the functionality defined in the Formal Design and that all tracing to the Formal Design is appropriate.
	INFORMED Design	To ensure that the functionality is implemented within the context of the architecture presented in the INFORMED Design.
	Coding Standard	To ensure code conforms to standard layout and language constraints. Also all static analysis warnings and errors are checked to ensure they are suitably justified and any un-discharged run-time error check VCs are justified.
Code Proof	Security Properties	To ensure that translation of security property to code proof context is valid.
	Formal Design	To ensure that translation of invariants and functional properties to the code proof context from the Z Design are valid. Also all proof rules introduced to discharge proofs using tool support are reviewed for correctness as are all justifications of VCs that are not discharged using tool support.
System Test Specification	Formal Design	To check that coverage of all possible schemas in the design is achieved by the system tests, that the expected test results match the functionality defined in the Formal Design, and that tracing from tests to the Formal Design is appropriate.

Table 1 Review Processes

3.1.10.2 Outputs

The outputs of this activity were:

- Review records for each review undertaken, these typically comprised annotated copies of the entity under review.

3.1.10.3 Key Benefits

- Review of each lifecycle phase against the previous provides a cost-effective check that errors have not been introduced when performing the new lifecycle phase.

3.1.10.4 General Applicability

On a larger project the review records would normally consist of a formal, itemised record of the review meeting. The reviews may also be attended by more people, provided always that each person at the review meeting has a well-defined role and brings a specific viewpoint to the meeting.

3.1.11 Fault Management

3.1.11.1 Approach

From the point that an output from a lifecycle phase is reviewed all updates to that entity are performed through the fault control process.

The fault control process at Praxis plays an important role in the development process. It captures all the failures found during and after the development activity and it provides a mechanism for ensuring that all faults are corrected and that all products of the development process (documents, source code, test scripts, etc.) are kept consistent.

When a potential failure is found it is logged, with a description of the problem. This description may range from a description of the observed system behaviour to a description of an error in a document. Once a failure has been logged it cannot be closed until a full evaluation has taken place.

Failure evaluation establishes the source cause of the problem, if any (ie the point in the lifecycle that the problem was introduced) and determines the entities that need modification to correct the fault and make all documents consistent.

The implementation of all corrections is then tracked through to review. Only when the implementation of the fault correction has been completed and reviewed can the fault be closed.

By analysing open faults we have a clear understanding of known outstanding problems. Typically we ensure that any delivered build has no known failures, or, if this is impractical, document the known failures at the time of release.

The fault management process was controlled by a simple spreadsheet that maintained the status of all failures and incident reports used to detail each failure. Incident reports were paper based. The whole fault reporting system was implemented with very simple technology and was not incorporated electronically into the configuration management tools used on the project. This was simply because of the tools chosen to support the project.

3.1.11.2 Outputs

The outputs of this activity were:

- Incident reports detailing all failures.
- A summary of all incident reports.

3.1.11.3 Key Benefits

The reasons for using a fault control process are:

- Fault management creates an audit trail for all changes required to any lifecycle entity once that entity has been initially approved.
- Fault management tracks useful statistics, such as when faults are found and when they are introduced, which can be used to monitor and improve process. If many faults are being introduced in a particular lifecycle phase then efforts can be made to improve the lifecycle phase in future projects.
- Fault management ensures that all issues are tracked and failures do not enter the delivered system without investigation.

3.2 Comparison with EAL5 and higher

The Common Criteria EAL5 Security Assurance Requirements are specified in part 3 of [12], as a list of codes and levels for each of the applicable assurance aspects. The following table summarises how the approach followed in this project meets (or in places exceeds) these requirements. Where the process exceeds the requirements for EAL5, the name of the requirement is *italicised*, the part of the implementation that exceeds the requirement is *italicised*, and the approximate level of requirement achieved is stated.

Configuration Management	ACM_AUT.1 Partial CM automation	Standard Praxis configuration management was carried out, using an automated tool to manage all the code.
	ACM_CAP.4 Generation support and acceptance procedures	Unique version numbering of the TOE and the design documents leading to the TOE. All configuration items (design documents, code, test scripts, etc.) have unique references.
	ACM_SCP.3 Development tools CM coverage	All documents developed as part of the TOE development are under configuration management with documented processes for creating, numbering, modifying, reviewing and issuing.
Delivery and Operation	ADO_DEL.2 Detection of modification	Not implemented: formal delivery to operational environment out of scope of project.

	ADO_IGS.1 Installation, generation, and start-up procedures	Installation and User manual documents all these procedures.
Development	*ADV_FSP.3 Semiformal functional specification*	A *formal* functional specification was provided. *(ADV_FSP.4)*
	ADV_HLD.3 Semiformal high-level design	A *formal* functional design specification was provided. In addition, a description of the interface to hardware specific aspects, and a breakdown into subsystems was provided in the INFORMED design. *(ADV_HLD.5)*
	ADV_IMP.2 Implementation of the TSF	The implementation is in *SPARK Ada, which is highly structured, and this structuring is enforced through static analysis tools. (ADV_IMP.3)*
	ADV_INT.1 Modularity	The implementation is in *SPARK Ada, which is highly structured, and this structuring is enforced through static analysis tools (ADV_INT.3)*
	ADV_LLD.1 Descriptive low-level design	Not implemented: the size and complexity of the TIS did not require this number of refinement levels. Indeed, our experience is that mandating a fixed number of refinement levels is counterproductive.
	ADV_RCR.2 Semiformal correspondence demonstration	Tracing carried out between all representation levels. *Formal proofs (partial)* of correspondence between functional specification and design specification. Full correspondence between design and code *(code annotations) and partial proof. (partial ADV_RCR.3)*
	ADV_SPM.3 Formal TOE security policy model	Formal policy model expressed as theorems on the formal functional specification.
Guidance documents	AGD_ADM.1 Administrator guidance	User guide covers administrator. True instructions to users and administrators for true secure operation are out of scope of this project.
	AGD_USR.1 User guidance	User guide covers administrator. True instructions to users and administrators for true secure operation are out of scope of this project.

Life cycle support	ALC_DVS.1 Identification of secure measures	References made to standard Praxis working practices. For this development, additional security measures were not deemed necessary.
	ALC_FLR.0 Flaw remediation	No requirements.
	ALC_LCD.2 Standardised lifecycle model	Praxis Correctness by Construction development process adopted. Industry standard in that it uses accepted technology (formality, refinement, SPARK Ada), but many of these approaches are regarded as novel by most developers.
	ALC_TAT.2 Compliance with implementation standards	*Standards used for all aspects of the development: REVEAL® (internal standard) for requirements, Spivey Issue 2 for Z notation, SPARK Ada (standard enforced by tools, unambiguous definition). (ALC_TAT.3)*
Tests	ATE_COV.2 Analysis of coverage	System test scripts derived from, and traced back to, the functional specification.
	ATE_DPT.2 Testing: low-level design	The standard Praxis approach of achieving 100% source code coverage through a combination of system testing, with additional unit tests introduced to cover areas difficult to test through system test, ensures that all internal interfaces are tested.
	ATE_FUN.1 Functional testing	System testing carried out, with full documentation.
	ATE_IND.2 Independent testing – sample	Independent reliability testing being carried out by SPRE Inc. *This may extend beyond the system testing carried out by Praxis. (ATE_IND.3)*
Vulnerability assessment	AVA_CCA.1 Covert channel analysis	Not applicable to this application.
	AVA_MSU.2 Validation of analysis (misuse)	Not implemented: True instructions to users and administrators for true secure operation are out of scope of this project.
	AVA_SOF.1 Strength of TOE security function evaluation	In general, the security mechanisms requiring analysis in this way are cryptographic features and aspects of securing access to data (such as securing access to the audit log). These were out of scope of this project.

	AVA_VLA.3 Moderately resistant (Vulnerability analysis)	In general, the security mechanisms requiring analysis in this way are cryptographic features and aspects of securing access to data (such as securing access to the audit log). These were out of scope of this project.

Table 2 Comparison with EAL requirements

3.3 Applicability

Although this project was a *demonstration* of a development approach, and it was necessarily small and focussed, the conclusions that can be drawn are scalable, for the following reasons:

- The structure of the documents, such as the functional specification, was designed to allow expansion. The full power of the formal notation's structure was used, and as a result the formal specification and formal design documents were larger and more complex than strictly necessary for the job in hand. But by ensuring that good structuring was used, these documents could easily be expanded to cater for additional functionality. For example, adding new administrator functions requires only that new function names are added, and new Z operations to describe the behaviour of each new function.

- The results (effort, fault-rates) found on this project are comparable with the results found on other, larger, non-demonstration projects that Praxis have carried out for other clients.

- Some activities were reduced in scope (such as requirements investigation, a number of the proof stages, and some of the security analysis), but this was only for cost reasons, and given a larger budget these could have been completed within the same project structure.

- Actual evaluation and certification would require external evaluators reviewing documentation and implementations. Although all review was carried out in-house, final reliability demonstration testing was carried out by an independent tester.

- Some simplifications were made on this project to focus effort on the core functionality. These were:

    - peripherals were simulated, rather than real
    - some library functions were simulated, rather than real
    - no graphical user interface was developed
    - the underlying operating system was not securely locked-down

However, the effect that these had on the development were lessened because

- peripherals were developed by an external supplier (SPRE Inc), and hence clear specifications were needed, and our development had to respond to unexpected alterations in the interfaces.

- experience on previous projects has shown us that it is possible to develop non-security-critical user interfaces in a conventional manner using prototyping and GUI builders, and then integrate them with the high-integrity development successfully.

- the issues of the underlying operating system and the application code can be successfully separated, and we have done this on other projects.

Praxis High Integrity System's development approach has been used on a number of projects for a range of clients. An example pertinent to this development is a project carried out for Mondex International (MXI), see [13].

4 Metrics

4.1 Raw Metrics

4.1.1 Effort Metrics

In order to allow detailed analysis of the effort involved in the various tasks undertaken an assessment of the competency levels of the engineers undertaking the work on this project was made.

Each engineer was assessed as a Novice, Practitioner or Expert (as defined in Table 3) in a number of skills key to the project.

Competency level	Definition
Novice	Has attended relevant training but has no experience in the given (or a closely related) activity.
Practitioner	Has attended relevant training and has sufficient experience in the given activity (or a closely related activity) to perform activity with minimal supervision.
Expert	Has several years experience and can supervise both Practitioner and Novice in the activity.

Table 3 Competency Levels

The key skills required during the lifecycle of the project were as follows:

- Requirements Elicitation
- Writing Z
- Z Proof
- Security
- INFORMED Design
- SPARK Coding
- Writing SPARK Proof Annotations
- SPARK Proof
- System Testing

The key technical skill required to undertake each of the technical WBS items was determined allowing full analysis of the skills required and the skill levels applied to the various activities in the development process.

When activities were undertaken an assessment was made as to whether the task was hard or easy. Tasks were classed as easy if a novice could perform them; while hard tasks were those where expert knowledge or experience was utilised. Review activities were classified as hard since it is preferential to have an expert perform the review. In many cases there were aspects of an activity that were hard while other aspects were easy. For example developing the type model and state schemas for the Z specification were hard but specifying many of the operations was classified as easy since, once one

operation was in place it could be used as a template for a novice to produce the specification of further operations.

The full raw effort metrics collected on the project are presented in full in Appendix B. They are summarised in the table below:

		Difficulty of Activity			Percentage of Effort
		Hard	Easy	N/A	
Lifecycle Phase	**Manage Project**	0	0	216	11%
	Define Requirements	68	124	0	10%
	Specify System	165	69	0	12%
	Design Core Functions	170	130	0	15%
	Code and Prove	104	453	0	29%
	System Test	0	76	0	4%
	Interfaces and Integration	0	316	0	16%
	Acceptance	10	53	0	3%

Table 4 Actual effort spent on lifecycle phases

Figures are in hours. Technical activities were classified as hard or easy. Project management was not classified as either hard or easy.

The main observations that can be drawn from this raw data are:

- Early lifecycle phases tended to require a higher proportion of expert involvement.
- Approximately 10% of the project activity was management. This percentage is commensurate with our experiences on other projects.
- The proportion of time spent on system testing was extremely low. A more representative figure would include the contribution from SPRE, including the production of the test environment (see Section 3.1.9.4). A more normal proportion from other Praxis projects is 25%.

4.1.2 Defect rates

During the development of the system all faults [1]detected following review of a deliverable item were captured using the fault management procedure. The severity of each fault was determined, in terms of the impact of the fault remaining in the system. Severity levels are defined in Table 5.

Severity Level	Definition
Critical	Failure is due to a Fault in the TIS Core and could compromise the security of the system.
Major	Failure is due to a Fault in the TIS Core and could impact the functionality of the system.
Minor	Failure is due to a Fault in the TIS Core but would not prevent the system from functioning (e.g. incorrect spelling of displayed text).
Interfaces	Failure is due to a fault that is confined to the support software; there is no fault in the TIS Core software.
Test	Failure is due to a fault in the test or test environment not the system under test.
No Fault	The reported failure is actually the correct behaviour so there is no fault.

Table 5 Failure Classifications

Table 6 summarises the distribution of failures found during development. Several failures were found to affect more than one item, typically this was where a fault was introduced early in the development lifecycle but not detected until relatively late in the development lifecycle.

[1] For the purposes of this document, a *failure* is an incorrect external behaviour of the system, and a *fault* is an error in any part of the system or its development products, which may or may not manifest itself as a failure of the system.

		Critical	Major	Minor	Interfaces	Test	No Fault	Total
Item Affected	**Requirements**	0	0	0	0	0	0	0
	Z Specification	0	6	6	0	0	0	12
	Security Specification	0	0	0	0	0	0	0
	Z Design	0	5	12	0	0	0	17
	INFORMED Design	0	6	8	0	0	0	14
	Code	0	10	17	0	0	0	27
	Interfaces	0	0	1	2	0	0	3
	System Tests	0	0	0	0	0	0	0
Total		**0**	**16**	**36**	**2**	**0**	**0**	**54**

Table 6 Distribution of failures by severity

The fault management process determines where faults are found and where they are introduced. This distribution is shown in Figure 5.

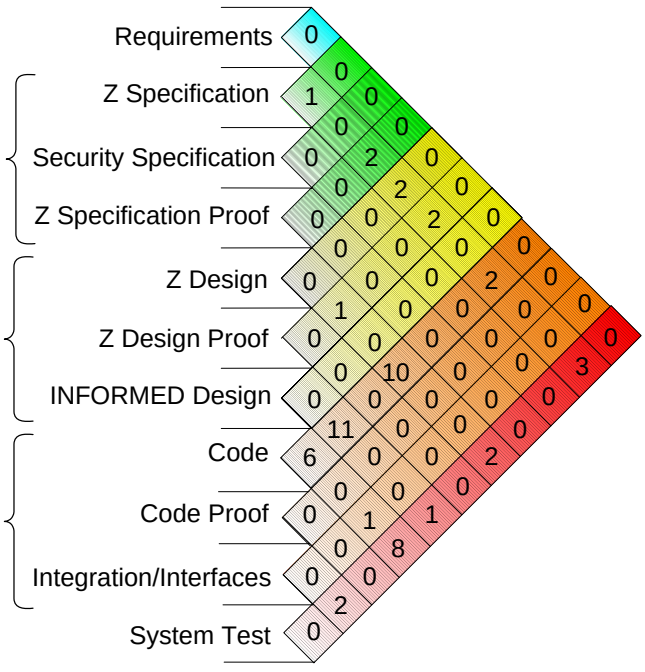

Figure 5: Faults found vs. point of introduction

Reading diagonally upward from a lifecycle phase gives the number of faults found during that lifecycle phase categorised by the point of introduction of the fault. Reading diagonally down gives the number of faults per artefact/lifecycle phase. So 8 faults were introduced during coding and found during system test, 6 faults were found and introduced during coding.

The correctness by construction approach aims to have as few defects on the right hand side of the triangle as possible. Ideally no defects would be introduced during the development process but realistically we attempt to eliminate defects as soon as possible resulting in faults being found soon after they are introduced.

4.1.3 Code Metrics

The code size is given in Table 7. The size of the source code was measured in terms of non-blank, non-comment lines. These were further categorised into declarations and executable lines of code. The non-blank comment lines were categorised into SPARK annotations (basic flow annotations and proof annotations) used during static analysis by the SPARK Examiner and simple textual comments. The total is a simple line count of all source code so includes blank lines.

Notice that the Support Software was written in Ada, rather than SPARK, so there were no SPARK annotations.

	Declarations	**Executable lines**	**SPARK flow annotations**	**SPARK proof annotations**	**Comments**	**Total**
TIS Core	4964	4975	6036	1999	8529	30278
Support Software	1800	1897	-	-	2240	6925

Table 7: Code sizes for Core and Support Software

4.1.4 Document Metrics

The characteristics of the deliverable documents are presented in Table 8.

Document	Reference	Form	Number of Pages
Project Plan	S.P1229.2.1	English text	27
Security Target	S.P1229.40.1	English text	18
Software Requirements Specification	S.P1229.41.1	English text	42
Formal Specification	S.P1229.41.2	Z Notation and English text	118
Security Properties	S.P1229.40.4	Z Notation and English text	11
Interface Specification	S.P1229.41.3	English text	84
Formal Design	S.P1229.50.1	Z Notation and English text	171
INFORMED Design	S.P1229.50.2	English text	67
Code Verification Summary	S.P1229.52.1	English text	23
System Test Specification	S.P1229.63.1	English text	98
Installation Guide and User Manual	S.P1229.73.1	English text	29

Table 8: Document Metrics

4.1.5 Code Proof Metrics

Code proof was broadly divided between

1 proving the absence of run-time errors, a general soundness property of the code, which ensures that there will be no run-time exceptions.

2 proving security properties were held by the code.

These two proof activities result in the SPARK Examiner generating a number of VCs (Verification Conditions) for each subprogram which need to be shown true to conclude the proof of the subprogram. The statistics from the code proof activity are shown in Table 9 and Table 10. VCs associated with assertions and pre and post conditions result from proving the security properties.

	Number
Subprograms fully proved automatically by Examiner/Simplifier	223
Subprograms fully proved by Checker	14
Subprograms fully proved by review	55
Subprograms for which VCs have been generated	292

Table 9: Proof mechanism used to fully prove each subprogram

	Total	**Proved by**		
		Examiner/ Simplifier	**Checker**	**Review**
Assert or Post-condition	1021	927	38	56
Precondition check	67	47	8	12
Check statement	1	0	0	1
Runtime check	1340	1293	2	45
Refinement VCs	214	186	9	19
Totals	2643	2453	57	133
Total %		93%	2%	5%

Table 10: Proof of VCs by Type

It should be noted that VCs that are proved by the Examiner/Simplifier require no manual intervention. Only those proved by the Checker or justified by review require manual intervention.

4.2 Summarised Metrics

4.2.1 Productivity rates

The following productivity rates can be deduced from the raw metrics:

4.2.1.1 Code productivity

The productivity rate for coding, accounting only for compiled lines of code (declarations and executable lines) is shown in Table 11. In calculating the overall productivity for the TIS Core all effort was considered. Figures for the TIS support software only consider the effort required for specifying and developing the interfaces.

Productivities are presented separately since the Correctness by Construction process was only applied to the TIS Core software.

	Productivity (LOC/day)	
	During coding	**Overall**
TIS Core	203	38
TIS Support	182	88

Table 11: Code productivity

Notice that the productivity during coding for the TIS core is higher than for the support software despite the core coding effort including static analysis. This is because there was very little rework of the TIS core software since the early lifecycle activities produced an unambiguous definition of the required software functionality.

4.2.1.2 Documentation productivity

The productivity rate for the production of each of the documents is shown in Table 12.

The lowest productivity was for Z proof arguments, this is because the effort in performing the necessary proof analysis far outweighs the time taken to document the process.

Productivity rates for documents written in the Z notation are typically lower than for documents written in English. The relatively high productivity rate for the Formal Design reflects the high level of reuse of structure that was possible from the Formal Specification.

Overall, the document productivity rate for this project is exceptionally high — productivity for Z specifications is more normally 1 – 2 pages per day. This high productivity is due partly to the quality and experience of the staff working on these areas, and partly to the small team size. With only a single

person writing a specification and one person reviewing, communications and confusions are kept to a minimum.

On the other hand, the specifications were unusually large given the complexity of the system. This was because they were deliberately constructed to be extendable, and more structuring was used that was warranted for the complexity actually captured.

For productivity figures for a larger project, see [13].

Document	**Number of Pages**	**Pages / day**
Security Target	18	2
Software Requirements Specification	42	11
Formal Specification	118	4.6
Security Properties	11	3.3
(Specification)	*7*	*7*
(Proof)	*4*	*1.7*
Interface Specification	84	3.9
Formal Design	171	6.9
(Design)	*153*	*8.3*
(Abstraction Relation)	*11*	*11*
(Proof)	*7*	*1.3*
INFORMED Design	67	5.7
System Test Specification	98	22
Installation Guide and User Manual	29	17

Table 12: Document productivity

4.2.1.3 Defect rates

Defect rates are typically quoted for a system post acceptance. We await results of reliability demonstration testing to give defect rates for the system.

4.2.2 Effectiveness of techniques

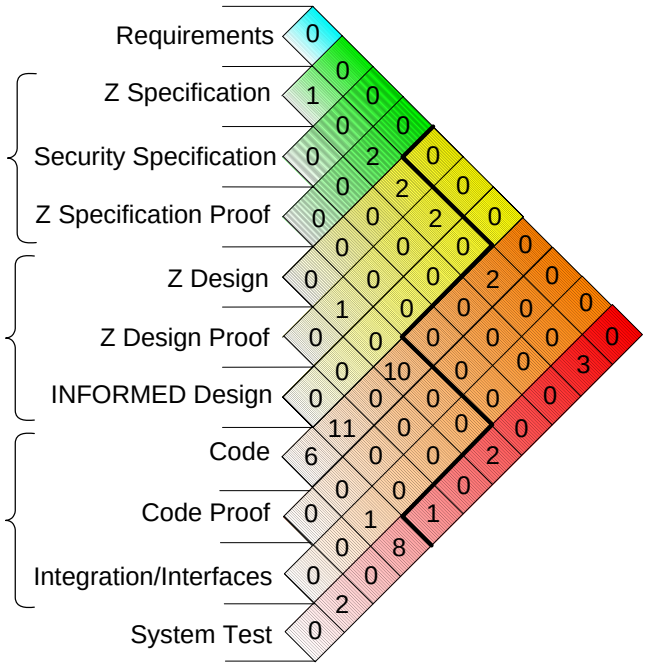

Figure 6: Faults found vs. point of introduction (repeated)

The heavy line down the triangle delimits where faults are considered to have been found in time as opposed to later than ideal.

In an ideal development the errors introduced at one stage of the development process are then uncovered either as part of that stage itself or, more likely, in the immediately following stage. In the style of the diagram in Figure 6 this would have values only in the first and second vertical columns.

As some of the "stages" in the TIS project are actually more-or-less concurrent, such as the Z Specification the Security Properties and the Z Specification Proof, we would expect errors to be identified in the area to the left of the heavy line shown in Figure 6.

Therefore, it is worth investigating in more detail the errors that were picked up later than ideal — to the right of the heavy line. There are eight of these, detailed in the subsections below. But in summary:

- 3 were errors with the User Interface, which would normally be picked up by UI prototyping (out of scope for this project)
- 1 error due to lack of reachability analysis for Z specifications
- 2 errors due to reduced requirements analysis and reduced requirements change tracking, adopted due to budget constraints and an expectation (mistaken) that the project was redeveloping functionality already fully defined.

- 2 errors due to moving between formal and informal representations that even the full high-integrity process would have missed.

4.2.2.1 Errors in Formal Specification found during System Test (three errors)

Incident Report no. 33: the initialisation state in the formal specification, although possible and secure, did not allow any useful subsequent use of the system. The timeouts and allowed access periods were unrealistic.

> This was not found during analysis of the Z specification because the normal analysis carried out on Z initialisation (proof of the existence of an initial state) misses these sorts of errors. Carrying out reachability analysis (not done here) would catch these sort of problems.

Incident Report no. 36: the console did not display a suitable “busy” message when it was busy.

> As the graphical user interface was out of scope of this project, only a simple console interface was built. Normally, during requirements analysis a user interface prototype would have been built, and this would almost certainly have found this error early on.

Incident Report no. 38: the console did not display a suitable message when an operation failed.

> As the graphical user interface was out of scope of this project, only a simple console interface was built. Normally, during requirements analysis a user interface prototype would have been built, and this would almost certainly have found this error early on.

4.2.2.2 Errors in Formal Design found during System Test (two errors)

Incident Report no. 34: an invariant on the configuration data was missing allowing useless Authorisation Certificates to be produced, and then denying access to the enclave.

> A more complete requirements analysis would probably have found this issue, or at least investigated the implications of the non-deterministic specification earlier, possibly identifying this as an allowed behaviour. Tighter management of change would also have identified this earlier, as the error was introduced when fast-track entry was developed.

Incident Report no. 35: the timeout of the Admin Token was not audited.

> Tighter management of change would probably have identified this at design time (when auditing was developed) — development ran ahead of client review, and the token timeout was added late.

4.2.2.3 Errors in Formal Specification found during Coding (two errors)

Incident Report no. 6: system statistics missed the removal of the User Token before processing finished.

> The step from the *informal* requirements specification (that all failures of entry should be recorded in the statistics) to the *formal* specification of the actual processing is always a weak link, and any process may have missed this.

Incident Report no. 32: poor choice of wording for message in the console.

> As the graphical user interface was out of scope of this project, only a simple console interface was built. Normally, during requirements analysis a user interface prototype would have been built, and this would almost certainly have found this error early on.

4.2.2.4 Errors in INFORMED Design found during System Test (one error)

Incident Report no. 37: constraints on Configuration data not presented in the definition of valid input.

> The constraints on configuration data in the formal design should have been carried through to the (informal) INFORMED design definition of valid input data. This was missed during review.

4.2.2.5 Other observations on defect detection

Returning to Figure 6 again we note the following:

- No faults were found in the requirements. This is most likely due to the requirements omitting much detail (for example a detailed analysis of behaviour following failure). The majority of the faults relate to detailed system behaviour. Consequently the first point at which faults appear to be introduced in during the Z Specification, the point at which the detail was first introduced.

- Code proof found no faults. The code proof activity only accounts for the proof of security properties. Proof of absence of run-time errors and static code analysis is done as part of the coding process so it is typically difficult to gauge the effectiveness of these static code verification activities. The performance of static analysis as part of the coding activity accounts for the relatively low number of faults found in the code during testing.

- Faults are only recorded against entities in the lifecycle that have undergone formal review and have been base-lined. Due to the relatively late availability of test drivers the interface code was base-lined after all other coding was complete. This accounts for the low number of faults reported on the interface software.

4.3 Interpretation of Metrics

The raw data presented corresponds to the actual development undertaken by Praxis. Due to available resources some of the activities in the Correctness by Construction process being demonstrated were not completed.

There were also a number of occasions where relatively easy activities were performed by experts.

In order to gauge the cost effectiveness and applicability of the processes presented on this project we have provided an interpretation of the raw metrics to estimate:

- The effort required if team skills were reduced so that experts performed only hard activities.
- The cost and effort required to complete the project if all the Correctness by Construction activities were completed.
- The cost and effort required to perform the project to EAL5 certification standard.

For the purpose of this analysis we exclude the Project Management aspect. Our experience is that project management typically accounts for about 10 - 15% of the effort and this is borne out in the raw effort metrics (see Section 4.1.1).

4.3.1 The Raw data

Raw data is repeated here to allow easy comparison

	Effort (hours)		
	Hard	**Easy**	**Total**
Define Requirements	68	124	192
Specify System	165	69	234
Design Core Functions	170	130	299
Code and Prove	104	453	557
System Test	0	76	76
Interfaces and Integration	0	316	316
Acceptance	10	53	63
Total	517	1219	1736

Table 13 Actual effort spent on lifecycle phases

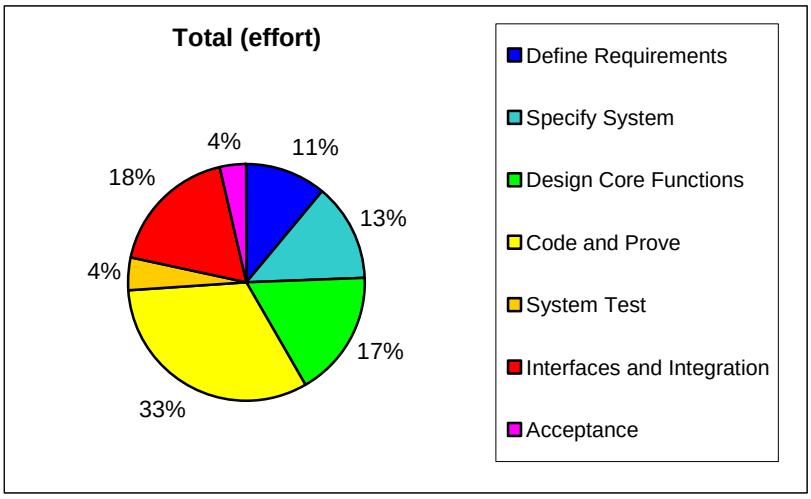

Figure 7: Effort distribution

4.3.2 Perfect team distribution

In an ideal team distribution all hard activities would be performed by experts in the field while all easy activities would be performed by novices. Of course, the novices would graduate to practitioners over the course of the project but this is not accounted for in the estimates presented here.

In order to analyse the likely time to complete if experts always perform hard activities and novices always perform easy activities we make the following assumptions about the effective productivity of the various skill levels. These assumed productivity ratios are equal to the cost ratios of staff at representative grading levels.

Skill level	Productivity
Novice	100%
Practitioner	165%
Expert	230%

Table 14 Productivity Rates

Productivity rates are presented as a percentage of Novice productivity.

This gives the following effort estimates for the work actually performed:

	Effort (hours)		
	Hard	**Easy**	**Total**
Define Requirements	54	239	293
Specify System	165	158	323
Design Core Functions	159	277	436
Code and Prove	88	914	1002
System Test	0	115	115
Interfaces and Integration	0	426	426
Acceptance	10	53	63
Total	476	2181	2658

Table 15 Effort spent on lifecycle phases assuming a perfect team

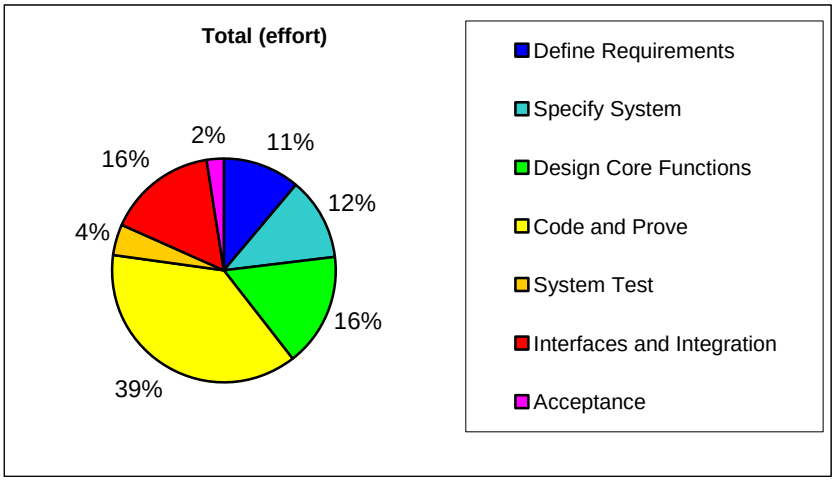

Figure 8: Effort distribution assuming a perfect team

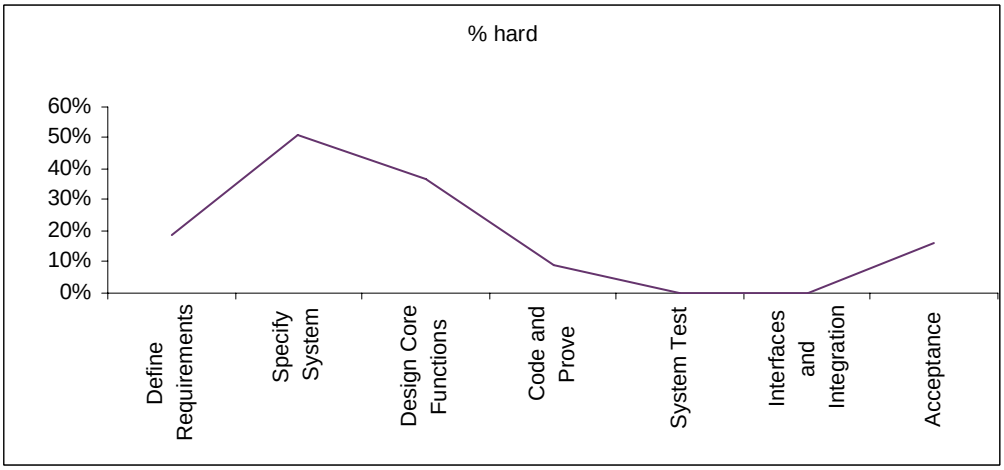

Figure 9: Proportion of lifecycle activities classified as hard

The key observation that can be drawn from this is that in general more expertise is required early in the lifecycle than during the later stages (hard activities require experts).

We now look at the cost impact of this distribution of experts and novices over the life time of the project, using the same cost ratio as productivity ratio (Expert costs 2.3 times the cost of a Novice) and consider the cost distribution over the lifetime of the project.

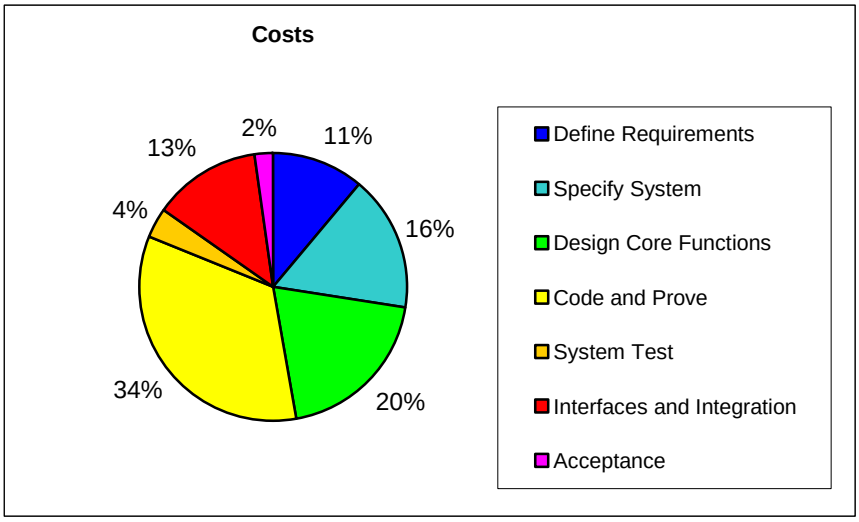

Figure 10: Cost distribution for work performed

It should be noticed that almost half of the project costs were incurred prior to coding. This reflects the emphasis placed on correct construction of requirements, specification and design.

4.3.3 Completing the Correctness by Construction Process

A number of the activities performed as part of the Praxis Correctness by Construction process were not completed due to budget limitations. Here we estimate the extra effort required to complete these activities and conclude with the effort and cost distribution that would be expected had all activities been completed. Note that this evaluation concerns just the core functionality of the system and in addition we would expect a contribution from the production of a test environment. The effort required for generating a test environment should be deduced from the effort required by SPRE to perform this activity and is not included in the figures below.

The activities that were not completed were as follows:

- **Requirements Elicitation** – restrictions on time available with the stakeholders interested in the system resulted in the requirements elicitation activity being far smaller than usual. We estimate that only 25% of the effort to complete was spent on this activity.

- **Write Software Requirements Specification** – due to reduced requirements elicitation, many aspects of the system behaviour that would have been elaborated in the requirements specification were only described at a very high level, detail being postponed to the Formal Specification. We estimate that only 50% of the effort to complete was spent on this activity.

- **Write Security Target** – We only produced an outline Security Target. We estimate that this activity took 40% of the effort to complete a Security Target.

- **Review SRS and ST** – The review activities will take proportionally longer due to the SRS and ST being larger documents if they had been completed. We estimate that we spent 50% of the effort that would have been required to review these documents.

- **Prove Security Properties** – We only proved a sample of the security properties. We estimate that this amounted to 30% of the effort required to complete the proof.

- **Prove Design** – We only performed part of the design proof, although more proof was actually performed than was documented. We estimate that this amounted to 40% of the effort required to complete the proof.

- **Review Design Proof** – We only reviewed the proof that was documented. We estimate that only 20% of the proof was documented.

- **Proof Annotations** – We produced approximately 30% of the proof annotations for the security properties. Due to the simple correspondence between the formal design and code full functional proof would probably not be performed in practice.

- **Proof of Code** – The effort required to prove the code is directly proportional to the number of proof annotations inserted. We proved all VCs associated with the annotations we supplied. This corresponds to 30% of the effort required to complete proof of all security properties.

- **Code Proof Review** – Again the effort expended corresponds to about 30% of the effort to complete.

- **Execute Functional Testing** - We would normally instrument functional tests to capture coverage metrics. We estimate that the testing we performed corresponds to about 50% of the effort that would have been required if the testing was instrumented and we added tests to achieve 100% code coverage as necessary.

- **Test Plan and Specs** – We wrote tests to cover all aspects of the Formal Design (and hence Formal Specification). We anticipate that additional tests would be required to achieve 100% code coverage (which was not monitored for). We estimate that we expended 60% of the effort required to complete this activity.

- **Test Report and Results** – Ordinarily we would have analysed the coverage metrics we captured. This would have added a substantial overhead to the collection and interpretation of test results. We estimate that we performed about 30% of the effort required to complete this activity.

Taking into account the above we estimate that the effort to complete the project (again assuming a perfect team) is as follows:

	Effort (hours)		
	Hard	**Easy**	**Total**
Define Requirements	151	585	736
Specify System	206	158	364
Design Core Functions	229	277	506
Code and Prove	185	1369	1554
System Test	0	233	233
Interfaces and Integration	0	426	426
Acceptance	10	53	63
Total	782	3100	3882

Table 16 Effort required to complete assuming a perfect team

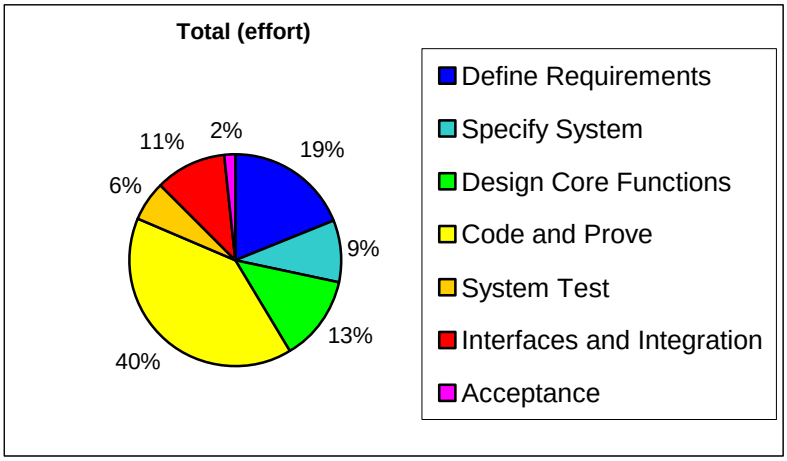

Figure 11: Effort distribution if Correctness by Construction process completed

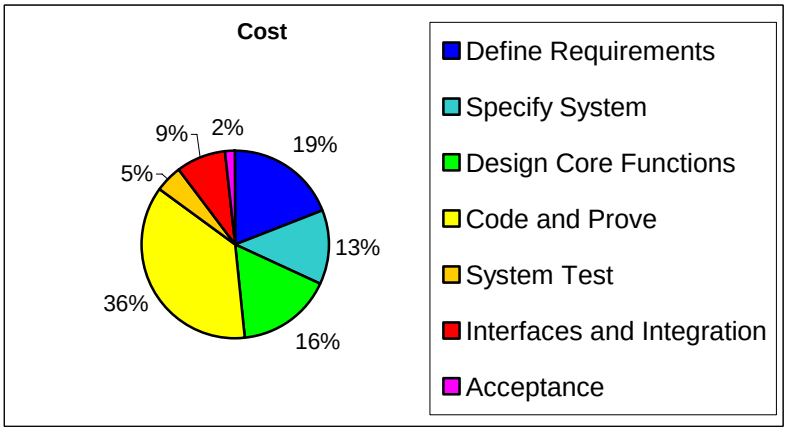

Figure 12: Cost distribution if Correctness by Construction process completed

4.3.4 Achieving EAL 5

A number of the activities performed in the Correctness by Construction process presented here are not actually required for EAL5. So assuming that we were able to justify with the certification authorities that the static analysis performed on the code removes the need to perform implementation testing we can deduce the effort and cost of developing this system to EAL5 by removing the effort for the following activities (not required for EAL5).

- Prove Security Properties
- Prove Design
- Review Design Proof
- Proof Annotations
- Proof of Code
- Code Proof Review

There would be additional effort required to write the installation guide, which needs to cover aspects of security not covered by the current installation guide. We assume that the effort required to produce this summary report is equivalent to the additional effort required to produce an installation guide suitable for EAL5.

These assumptions give an estimate of the effort required to develop this system to EAL5 (assuming a perfect team) as follows:

	Effort (hours)		
	Hard	**Easy**	**Total**
Define Requirements	151	585	736
Specify System	148	158	306
Design Core Functions	94	277	370
Code and Prove	47	719	766
System Test	0	233	233
Interfaces and Integration	0	426	426
Acceptance	10	53	63
Total	449	2450	2900

Table 17 Effort required to complete to EAL5 assuming a perfect team

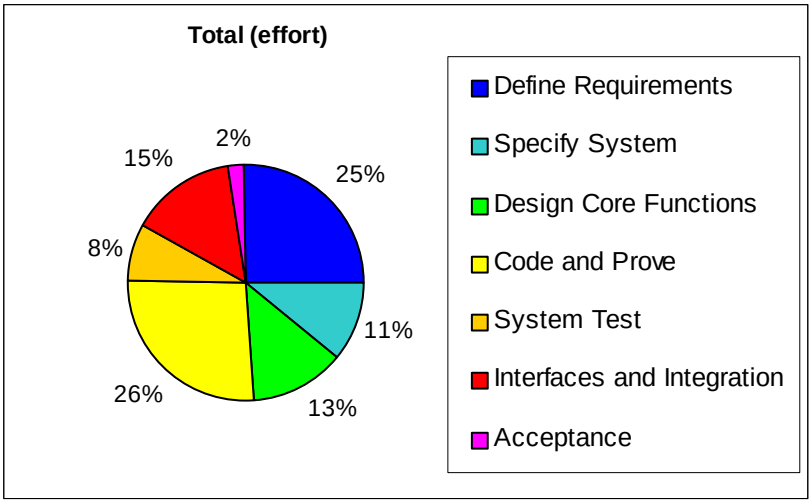

Figure 13: Effort distribution for completing to EAL5

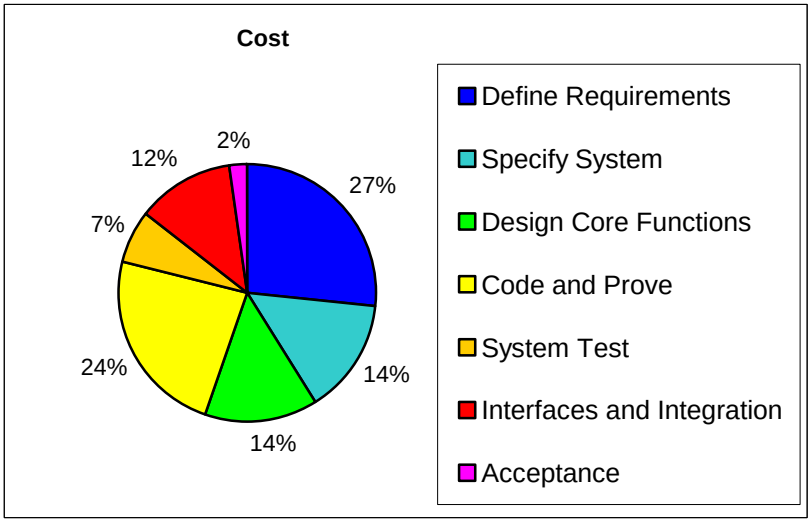

Figure 14: Cost distribution for completing to EAL5

4.3.5 Relative costs

The relative costs of the work performed, completing the Correctness by Construction process and completing to EAL5 are presented graphically

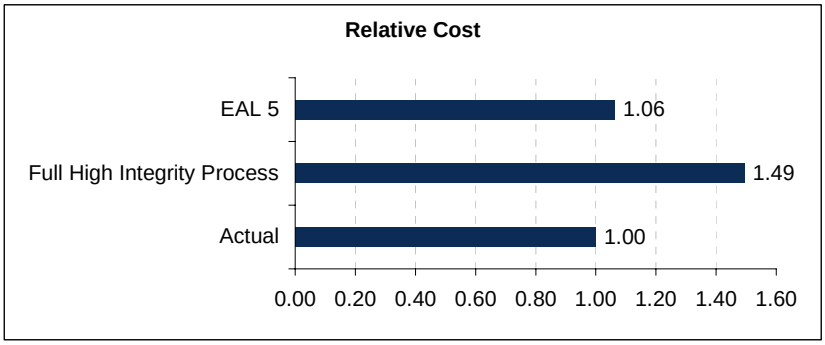

Figure 15: Relative costs of producing TIS core

Achieving EAL5 should cost little more than the actual costs measured during this evaluation. This is because, although we did not complete all activities within the Correctness by Construction process that we propose, many of the proof activities are not required to achieve EAL5 certification.

We believe that the Full Correctness by Construction process is close to achieving EAL7. There are a number of quality activities such as fault management and configuration management that would need to be performed using tools that provide fully integrated electronic support. The proof activity we use in our high integrity process is sufficient for EAL7: this involves tool supported code proof but manual proof of the Specification and Design.

5 Analysis

5.1 Analysis of Method

The metrics of effort expended in the phases of the project, the size of the system developed, and the number of residual faults found during reliability demonstration testing *(final metric awaited)* show that the Praxis Correctness by Construction development process as carried out in this project is effective in producing a high quality system cost effectively.

What are the elements of the method that achieves these benefits?

As indicated by the metrics relating number of errors found to phase in the development process where the errors were found shows that errors were introduced and found in all phases, and that no one phase can be pinpointed as the "key" place where errors were found. Indeed, part of the reason why, for example, coding introduced few errors is that the specification is formal, and hence clear and easy to implement from. The method brings together a number of techniques that work in unison to improve the quality of the product.

Having said that, the method does not rely entirely on *specific* techniques, such as Z or SPARK Ada. It will be possible to gain some of the benefit of the Praxis development approach even if only some of the elements are used, or if different techniques are used to achieve the steps. Our experience is that the approach adopted by us in this development achieves the greatest benefits, but if other factors force a departure from the ideal, then some benefit can still be gained. Looking at each part of the development in turn:

- Requirements management: Praxis' own REVEAL® requirements process stresses a number of elements of requirements management, and provides a number of techniques to address them. The key outcomes are: clear system boundary; clear stakeholder involvement; full investigation and documentation of system requirements; analysis and documentation of domain knowledge; and justification of system specification in terms of system requirements.

- Specification: formality is used to achieve clarity of expression, to force early expression of precise behaviour, and to allow more powerful verification and validation techniques to be applied. We have found that Z is a powerful, general-purpose formal notation, but other notations can be used. We have, for example, used CSP in particular circumstances. If a non-formal notation, such as UML, is used, then there is significant scope for ambiguity and lack of precision, and the opportunities for analysis are reduced (although even here, rigour in the use of UML can improve results [16]).

- Design: this is probably the area with the greatest amount of flexibility. The design step(s) are there to identify, document, and justify the implementation decisions made in passing from the abstract system specification to the concrete implementation. Different notations have different strengths for this, and the correct notation should be chosen to give the greatest power in expressing the design decisions being made, and the greatest scope for verification and validation techniques.

- Coding: choosing to implement the core of TIS in SPARK Ada gave the project the opportunity to apply powerful static analysis tools (the SPARK Ada tool-set) that eliminate many errors prior to compilation, and encourage (and measure) good design structuring. Other implementation languages can be chosen, but verification and validation power is lost, and poor design becomes more likely. Alternatives, such as using SPARK Ada as a design language with automatic translation into an implementation language, such as C, can be considered.

- Analysis: choosing appropriate formal specification and design notations, and SPARK Ada for implementation, allowed effective analysis:

  — review becomes very powerful, as the notations used are unambiguous, and truly abstract specifications allow useful correspondence checking down through progressive refinements

  — proofs of consistency of individual stages (e.g. pre-conditions checks) ensure that specifications are not only syntactically correct but have good meaning

  — proofs can be used to show correspondence between representation levels, right down to the code level

  — static analysis of the code can identify code-level errors (such as the use of un-initialised variables) and, with the use of formal notations in more abstract representations, can identify mismatches between specification and code.

- Testing: if static analysis, proof, etc. is carried out, this will remove most of the unit-level errors before code is run. Therefore, most of the errors remaining at runtime will be integration errors, to be identified through system testing. Once again, good specifications higher up the abstraction hierarchy lead to clearer test specifications and easier coverage analysis.

5.2 Analysis of Results

5.2.1 What the results means for EAL5.

As tabulated in section 3.2, this development project has largely met the requirements of EAL5, and in many cases, EAL6 and EAL7. As discussed in section 4.3.4, even the effort estimates for a fully-conformant EAL5 development are very close to the effort expended on this project. It is therefore reasonable to say that this project represents a close match to the process that should be followed for an actual EAL5 development.

However, issues of certification such as working with external evaluators and gaining agreement with the certification bodies have not been investigated in this project. Very little time was given to the development of the *Security Target,* and the development of the *Protection Profile* was out of scope. Our general experience, however, indicates that there are elements that could be done differently to make certification easier. In general:

- Engage with the accreditors and evaluators early, and ensure that they agree with all procedures and documents as you go.
- Ruthlessly trim the security target down to the fundamental security requirements.
- Develop specific security mechanisms to address specific threats, rather than incorporating a range of common mechanisms that have been proved to be useful in the past, but for which no clear justification can be identified for this specific product or system.
- Make trade-offs *at the security target stage* between technical security mechanisms, environmental security assumptions, and the effort of implementation. That is, analyse the threats created by the chosen environment, decide whether the implementation cost of protecting against these threats using technical means is acceptable, and if not, modify the environment to reduce the threats.

The consequences of adopting such an approach for TIS in particular would be:

- No protection profile (just develop a security target directly).
- A smaller security target (at least, the security target would be smaller than the current combination of the security target and the protection profile).
- Clearly recognised, high-level security aims, such as “protection of the enclave”.
- Clear reliance on the environment, making it easier to assess the impact of deploying in a different environment.

5.2.2 What the results means for EAL6/7

We believe that the Full Correctness by Construction process is close to achieving EAL7. The discussion above for achieving EAL5 remains valid at the higher EALs as well. It can be seen from section 3.2 that in many areas the project actually met the requirements of EAL6 or EAL7. If all of the proofs were carried out to completion, rather than only a sample completed, then the comparison of the project to EAL7 requirements falls into three areas:

- most requirements are met: the development requirements (on design documents, demonstration of correspondence between them) are met because formality was used throughout the development, and static analysis and proof was used where possible. Some testing and some lifecycle requirements are met.

- a few requirements are not met: these are generally process requirements (such as automated CM) that do not fundamentally alter the design process, but just require slightly tighter control on the activities that are already being done, or they are testing requirements, which in general are either superseded by static analysis or would normally be complied with on a project with a larger team.

- those requirements deemed out of scope: would need to be carried out, and would require additional effort.

5.2.3 What the results mean for achieving lower EAL levels

Even if a lower level of assurance is aimed for, the Correctness by Construction development process followed on this project can be used, and will yield a high quality system cost effectively. Indeed, provided there is a desire to reduce the number of residual errors sufficiently far, most of the processes adopted in this development represent the *most effective* way of developing the system.

There is a cut-off point in terms of quality at which this development processes ceases to be cost effective. If a high number of errors can be tolerated (e.g. in a system in use for limited time or by a limited number of people) then a less formal and rigorous process may be cheaper. The level of this cut off is not known, however.

6 Further Work

NSA's original purpose for this work was as a first step in improving the development of systems by NSA contractors toward high security certification levels. Having demonstrated that Praxis' Correctness by Construction development process can produce reliable systems that should be certifiable at the high EAL levels, it is worth considering the possible next steps toward NSA's final goal of wider contractor abilities to achieve these levels.

6.1 Disseminating the results of the project

There are a number of routes open to disseminate the results of this project to a wider audience, falling into three main areas: conferences, journals, and NSA-sponsored communications.

There are two conferences that naturally lend themselves to presentations about this project: NSA's own annual conference in April, and the Common Criteria international conference (probably September 2004).

There may be scope for introducing some of the techniques used on this project into NSA's own National Cryptologic School. Initial aspects to discuss are probably REVEAL® (requirements management), SPARK (design, implementation, and code), Z (formal specification and formal design), and a more general process view, covering the whole lifecycle.

As the NIAP labs will have experience of independent assurance, a more interactive discussion with their representatives may be fruitful, sponsored by NSA.

We would like to aim toward journal publication, also, and would be happy to discuss joint authorship.

6.2 Raising the development capabilities of contractors

To achieve the long-term aim of improving the take up of Common Criteria as a certification mechanism, the lessons learnt in this experimental development will have to alter the development practices of the majority of the NSA's contractors. This in turn will need the contractors to pass through four phases: understanding, belief, learning, experience.

- *understanding*
  The contractors need to be *exposed* to the principles of the Common Criteria, the lessons learnt during this project, and the concepts of the Praxis Correctness by Construction development process. This will require a continuing, wide-ranging dissemination activity.

- *belief*
  Having understood the demands of the Common Criteria and the promise of a development activity that can achieve certification, the contractors need to be convinced that the development process will work. They need to come to the belief that there is a business benefit in them adopting a new approach. This is a hearts-and-minds activity.

- *learning*
  For contractors to change their processes, they will have to learn new skills, define new processes, and work in a new way. This will require training courses, seminars and consultancy to transfer knowledge to the contractors.

- *experience*
  Training is never enough to sustain a change in a company. A continuing programme of change management is needed, with mentoring, coaching, advanced technique seminars, and support.

7 Conclusions

The TIS development project has demonstrated that the Praxis Correctness by Construction development process is capable to producing a high quality, low defect system in a cost effective manner following a process that conforms to the Common Criteria EAL5 requirements.

The TIS system's key statistics are:

- lines of code: 9939
- total effort (days): 260
- productivity (lines of code per day, overall): 38
- productivity (lines of code per day, coding phase): 203
- defects (defects found post delivery per 1000 lines of code): not currently known

The development approach applied on this project, and described in this report, is Praxis High Integrity System's standard high-integrity development process, and has been applied successfully to a number of commercial and government projects by Praxis. It is not new or under development — it is a proven technology. It has been shown to work on information processing systems, interactive systems, and real-time systems. Our experience in working with other system developers is that our development approach can be applied successfully by other companies, but the learning curve for many organisations is steep. Good training, a continuing mentor and coaching programme, and commitment to improvement are necessary to ensure that take-up of the approach is successful.

A Summary of the Behavioural Requirements

The required behaviour of TIS was described in the System Requirements Specification [1] in terms of a number of scenarios and some more general information.

A.1 Scenarios

The scenarios are summarised below (for brevity only the key assumptions and end conditions are listed here, failure conditions are not elaborated here, and nor is the list of audited events).

1 **User gains allowed initial access to Enclave**

Description: A user who is allowed access to the enclave is given access, making use of biometric authentication.

Stimulus: The user supplies their token to TIS via the token reader.

Assumptions:

- TIS is quiescent (no other access attempts, configuration changes or start-up activities are in progress).
- The user's token is valid and the I&A data on the token includes a valid fingerprint template that matches the fingerprint of the User's finger.
- The User is outside the Enclave; the door is closed and locked.
- The User's token does not have a valid, current Authorisation Certificate.

Success End-Conditions:

- The User is in the Enclave; the door is closed and locked.
- The User token contains a current, valid Authorisation Certificate.

2 **User is denied prohibited initial access to Enclave**

Description: A user who should not be allowed access to the enclave is prohibited access, possibly making use of biometric authentication.

Stimulus: The user supplies their token to TIS via the token reader.

Assumptions:

- TIS is quiescent.
- The User is outside the Enclave; the door is closed and locked.
- At least one of the certificates on the User's token is invalid, **or** the I&A data on the token does not include a valid fingerprint template that matches the fingerprint of the User's finger.
- The User's token does not have a valid, current Authorisation Certificate.

Success End-Conditions:

- The User is outside the Enclave; the door is closed and locked.
- The User token is unmodified.

3 **User gains allowed repeat access to Enclave**

Description: A user who should be allowed access to the enclave, is given access, but does not use biometric authentication because an Authorisation Certificate is found that is still within its validity period.

Stimulus: The user supplies their token to TIS via the token reader.

Assumptions:

— TIS is quiescent.
— The User is outside the Enclave; the door is closed and locked.
— The User's token has a valid, current Authorisation Certificate.

Success End-Conditions:

— The User is in the Enclave; the door is closed and locked.
— The User token is unmodified.

4 **ID Station is started and enrolled with input from the Enrolment Station**

Description: A person powers up the ID Station system, and loads the initialisation data from the Enrolment Station via a floppy disk.

Stimulus: Launching the ID Station application from the Windows Interface.

Assumptions:

— Enrolment data for the ID station is unavailable internally to the system.
— A floppy disk has been inserted into the drive, and the data on the floppy disk from the Enrolment Station is correct.

Success End-Conditions:

— The ID Station is running and ready for use, with the data as supplied from the floppy.

5 **ID Station is started already enrolled**

Description: A person powers up the ID Station system, and the ID station becomes available for use, as it has previously been enrolled.

Stimulus: Launching the ID Station application from the Windows Interface.

Assumptions:

— Enrolment data for the ID station is available internally to the system.

Success End-Conditions:

— The ID Station is running and ready for use.

6 **ID Station is shutdown**

Description: A Security Officer powers down the ID Station system.

Stimulus: Command to shutdown is typed into the console.

Assumptions:

— A Security Officer is currently logged onto the ID Station.

Success End-Conditions:

— The ID Station is no longer running and responds to no inputs.

7 **Security Officer updates the configuration of the ID Station**

Description: A Security Officer updates the ID Station configuration data with a completely new set of data, from a floppy.

Stimulus: Command to re-configure is typed into the console.

Assumptions:

— A Security Officer is currently logged onto the ID Station.

Success End-Conditions:

— The ID Station is available for use with its configuration identical to that specified on the floppy.

8 **Audit log is archived**

Description: An Auditor archives the audit log off the system onto a floppy disk, clearing the audit log on the ID Station.

Stimulus: Command to archive the log is typed into the console.

Assumptions:

— An authorised Auditor is logged on.

Success End-Conditions:

— The audit log on the ID Station no longer contains those audit elements that are now on the floppy disk.

— The oldest part of the audit log on the ID Station at the beginning of this scenario is on the floppy disk.

9 **Guard manually unlocks the door**

Description: A Guard overrides the latching and requests the door to be unlocked manually to allow the entry of a Person.

Stimulus: Command to unlock the door is typed into the console.

Assumptions:

— The ID Station is quiescent.

— The Guard is logged on.

— The User is outside the enclave; the door is closed and locked.

Success End-Conditions:

— The User is in the Enclave, the door is closed and locked.

10 **Administrator logs on**

Description: An Administrator logs onto the ID Station by inserting their Token in the Admin Token Reader.

Stimulus: A Token is inserted in the Admin Token Reader.

Assumptions:

— The ID Station is quiescent.

— The card inserted by the Administrator has a valid Authorisation Certificate.

Success End-Conditions:

— The ID Station is available for use by the Administrator, in that it will respond to the commands allowed to that Administrator as defined by the privileges in the Authorisation Certificate read from the Token and the Configuration data held on the ID Station.

11 **Administrator logs off**
Description: An Administrator logs off the ID Station.
Stimulus: The Token is removed from the Admin Token Reader.
Assumptions:
— An Administrator is logged on (which implies an Admin Token is in the Reader).
Success End-Conditions:
— The ID Station is unavailable for use by anyone at the console; it will respond to no commands typed in at the console.

A.2 General conditions

A.2.1 Audit Failure

The audit file should be a record of all auditable events that have occurred within the ID Station System. There are two distinct failures associated with the audit:

— Failure to write an auditable event to the audit file. Result: the Door is locked and the system shutdown.

— Space for audit files has been exhausted. Result: the oldest records are overwritten with the new audit records, and an alarm is raised to the Guard.

A.2.2 Doors and Alarms

The door can be open or closed; and can also be locked or unlocked. These two bi-state conditions are independent. In addition there is an alarming state: if the door is secure, then the alarm is silent. If the door is potentially insecure (it is open but locked, waiting for a user to pass through before closing the door and becoming secure) then the alarm is silent, but waiting a timeout period before alarming. If the timeout period passes, the alarm goes off.

As unlocked states are potentially insecure, there is always a time-out period, after which the door will be commanded to lock.

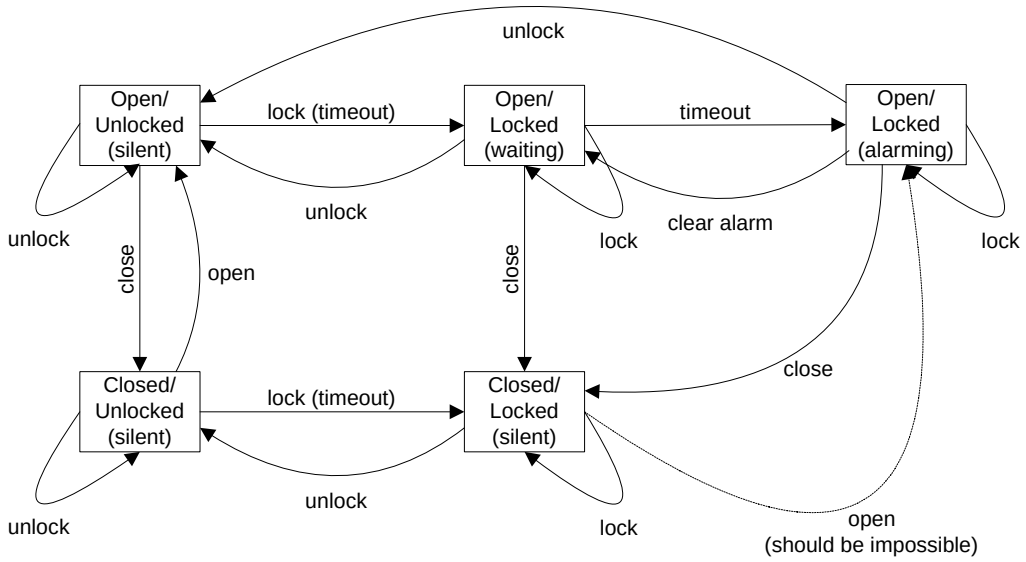

Figure 16: Open/Closed, Locked/Unlocked, and Alarm relationships

B Raw Effort Metrics

WBS #	WBS Descriptions	Key Technical Skill	By skill level and complexity: Novice Hard	Novice Easy	Practitioner Hard	Practitioner Easy	Expert Hard	Expert Easy	NA Hard	NA Easy	NA	Total
All entries are hours unless stated otherwise												
0	**Complete Project**		**62.50**	**69.50**	**18.00**	**393.00**	**426.25**	**540.80**	**10.00**	**216.00**	**215.50**	**1951.55**
1000	**Manage Project**										**215.50**	**215.50**
1100	Plan Project										115.00	115.00
1200	Reporting Project										80.50	80.50
1300	Supporting infrastructure										20.00	20.00
1400	Close Project											
2000	**Define Requirements**		**22.00**	**36.50**	**5.50**		**40.50**	**87.80**				**192.30**
2100	Study documents	Requ Elicitation	22.00	14.00								36.00
2200	Elicit requirements	Requ Elicitation		22.50			26.50					49.00
2300	Write SRS	Requ Elicitation					7.50	22.00				29.50
2500	Write security target	Security						65.80				65.80
2600	Review SRS and ST	Security			5.50		6.50					12.00
3000	**Specify System**						**165.00**	**68.50**				**233.50**
3100	Specify core functions	Writing Z					133.00	61.00				194.00
3200	Specify security properties	Writing Z						7.50				7.50
3300	Prove security properties	Z Proof					17.50					17.50
3400	Review spec and properties	Writing Z					14.50					14.50
4000	**Design Core Functions**		**19.00**	**17.00**			**150.75**	**112.50**				**299.25**
4100	Formal design	Writing Z					70.00	68.00				138.00
4200	INFORMED design	INFORMED Design	19.00	17.00			12.00	40.50				88.50
4300	Abstraction relation	Writing Z					3.50	4.00				7.50
4400	Review design	Writing Z					21.75					21.75
4500	Prove design	Z Proof					41.50					41.50
4600	Review proof	Z Proof					2.00					2.00
5000	**Code and Prove**		**21.50**		**12.50**	**197.50**	**70.00**	**255.00**				**556.50**
5100	Code	SPARK Coding				197.50		170.50				368.00
5200	Proof Annotations	Proof Annotations					15.00	15.00				30.00
5300	Proof of code	SPARK Proof					19.00	69.50				88.50
5400	Code review	SPARK Coding	21.50		12.50		28.50					62.50
5500	Proof review	SPARK Proof					7.50					7.50
6000	**System Test**			**16.00**		**60.00**						**76.00**
6100	Test plan and specs	System Testing				33.50						33.50
6200	Execute Functional testing	System Testing				26.50						26.50
6300	Test report and results	System Testing		16.00								16.00
7000	**Interfaces and Integration**					**135.50**		**17.00**		**163.00**		**315.50**
7100	Interface specification									161.50		161.50
7200	Interface implementation	SPARK Coding				135.50		17.00				152.50
7400	Integration testing									1.50		1.50
8000	**Acceptance**								**10.00**	**53.00**		**63.00**
8100	Write summary report								10.00	29.50		39.50
8200	Review summary report											
8300	Write installation guide									12.50		12.50
8400	Support reliability testing									11.00		11.00
	Totals (hours)		***62.50***	***69.50***	***18.00***	***393.00***	***426.25***	***540.80***	***10.00***	***216.00***	***215.50***	***1,951.55***
	Totals (days)		**8.33**	**9.27**	**2.40**	**52.40**	**56.83**	**72.11**	**1.33**	**28.80**	**28.73**	**260.21**

Table 18 Raw effort metrics gathered during development

Each engineer was assigned a competency level for each of the skills required. If no specific specialist skill was required then the competency is not applicable although tasks may still be recorded as hard or easy. The difficulty of project management activities was not classified.

Document Control and References

Changes history

Issue 0.1 (17 September 2003): Initial Draft proposing general structure.

Issuer 0.2 (6 October 2003) Proposed general structure following internal review. Supplied to NSA for comment.

Issue 0.3 (28 November 2003): first draft of content for internal review.

Issue 1.0 (17 December 2003): Provisional issue for client review.

Issue 1.1 (19 August 2008): Updated for public release.

Changes forecast

Updates following comments from NSA and completion of reliability demonstration testing.

Document references

1 TIS System Requirements Specification, Praxis High Integrity Systems Ltd, S.P1229.41.1.

2 TIS Formal Specification, Praxis High Integrity Systems Ltd, S.P1229.41.2.

3 TIS Security Target, Praxis High Integrity Systems Ltd, S.P1229.40.1

4 TIS Security Properties, Praxis High Integrity Systems Ltd, S.P1229.40.4

5 TIS Formal Design, Praxis High Integrity Systems Ltd, S.P1229.50.1.

6 TIS INFORMED Design, Praxis High Integrity Systems Ltd, S.P1229.50.2

7 TIS Code Verification Summary, Praxis High Integrity Systems Ltd, S.P1229.52.1

8 System Test Specification, S.P1229.63.1

9 TOKENEER User Authentication Techniques Using Public Key Certificates, Part 3: An Example Implementation, NSA Central Security Service INFOSEC Engineering, v1.0, 10 February 1998.

10 Statement of Work for TIS re-development, NSA, 27 September 2002.

11 J. Michael Spivey, The fuzz Manual, Computer Science Consultancy, ftp://ftp.comlab.ox.ac.uk/pub/Zforum/fuzz

12 ISO 15408, Common Criteria for Information Technology Security Evaluation, August 1999 (Version 2.1)

13 Correctness By Construction: Developing a Commercial Secure System, Anthony Hall and Roderick Chapman, IEEE Software, Jan/Feb 2002, pp18-25

14 Web site for SPARK: http://www.sparkada.com/

15 Is Proof More Cost Effective Than Testing?, Steve King, Jonathan Hammond, Rod Chapman and Andy Pryor, IEEE Transactions on Software Engineering, Volume 26 Number 8

16 High Integrity Ada in a UML and C World, Peter Amey, Neil White, Praxis High Integrity Systems Ltd. *submitted to Ada Europe 2004.*