Software Engineering
Correctness by Construction

Issue: 1.1
Status: Provisional
13th January 2004

Originator

Anthony Hall, Rod Chapman

Approver

Martin Croxford

Copies to:

SEI
Noopur Davies

Praxis Critical Systems
Dewi Daniels

Management Summary

Correctness by Construction is a radical, effective and economical method of building software with high integrity for security-critical and safety-critical applications. Praxis Critical Systems use it to produce software with extremely low defect rates – fewer than 0.1 defects per thousand lines of code – with good productivity – up to around 30 lines of code per day.

The principles of Correctness by Construction are:

1 Don't introduce errors in the first place.

2 Remove any errors as close as possible to the point that they are introduced.

These are achieved by

1 Using a sound, formal, notation for all deliverables. For example, we use Z for writing software specifications, so it is impossible to be ambiguous. We code in SPARK, so it is impossible to introduce errors such as buffer overflow.

2 Using strong, tool-supported methods to validate each deliverable. For example we carry out proofs of formal specifications and static analysis of code. This is only possible because we use formal notations.

3 Carrying out small steps and validating the deliverable from each step. For example, we develop a software specification as an elaboration of the user requirements, and check that it is correct before writing code. We build the system in small increments, and check that each increment behaves correctly.

4 Saying things only once. For example, we produce a software specification, which says what the software will do, and a design, which says how it will be structured. The design does not repeat any information in the specification, and the two can be produced in parallel.

5 Designing software that's easy to validate. We write simple code that directly reflects the specification, and test it using tests derived systematically from that specification.

6 Doing the hard things first. For example we produce early prototypes to test out difficult design issues or key user interfaces.

As a result, Correctness by Construction is both effective and economical:

1 Defects are removed early in the process when changes are cheap. Testing becomes a confirmation that the software works, rather than the point at which it must be debugged.

2 Evidence needed for certification is produced naturally as a by-product of the process.

3 Early iterations produce software that carries out useful functions and builds confidence in the project.

Contents

1 Introduction

This document describes Correctness by Construction, the Praxis Critical Systems process for developing high integrity software. This is a flexible process which we have used to develop security-critical and safety-critical software. It delivers software with very low defect rates, by rigorously eliminating defects at the earliest possible stage of the process. It is an economical process because the time spent on early deliverables is more than recouped in the very small amount of rework necessary at late stages of the project.

The process consists of a number of steps each producing a deliverable, supported by a number of generic activities such as configuration management. The process is flexible in that the techniques used for each step can vary according to the project, and the timing and extent of steps can be changed according to the needs of the application. However, all variants of the process are based on the strong principle that each step should serve a clear purpose and be carried out using the most rigorous techniques available that match the particular problem. In particular we almost always use formal methods to specify behavioural, security and safety properties of the software, since only by using formality can we achieve the necessary precision.

Section 2 is an overview of the process and describes its main characteristics. Section 3 gives more detail of the process steps. Section 4 describes generic activities that take place throughout the process. Section 5 gives examples of process use. Appendix A describes SPARK, a language designed for secure and safe systems development.

2 Overview of the Process

2.1 Process Outline

Figure 1 is a simplified diagram of the process. It uses the symbols shown in Figure 2. It shows the main activities and deliverables, and the general flow of time from top to bottom. It does not show some crucial aspects of the process:

1 There is more overlap between different activities than can be shown in a figure.

2 The figure omits the outputs of the validation steps. Any validation step can affect any previous deliverable and cause re-entry to any previous activity.

3 We build the system top down and incrementally.

Correctness by construction depends on knowing what the system needs to do and being sure that it does it. The first step, therefore, is to develop a clear statement of requirements. However, it is impossible to develop code reliably from requirements: the semantic gap is just too big. We therefore use a sequence of intermediate descriptions of the system to progress in tractable, verifiable steps from the user-oriented requirements to the system-oriented code. At each step we typically have several different descriptions of different aspects of the system. We ensure that these descriptions are consistent with each other and we ensure that they are correct with respect to the earlier descriptions.

1 The **User Requirements** describe the purpose of the software, the functions it must provide and the non-functional requirements such as security, safety and performance.

2 The **Software Specification** is a complete and precise description of the behaviour of the software viewed as a black box. It contains no information about the software's internal structure.

3 The **High Level Design** describes the architecture of the software.

4 A number of **Detailed Designs** describe the operation of different aspects of the software, such as its process structure or database schema.

5 **Module Specifications** define the state and behaviour encapsulated by each software module.

6 **Code** is the executable code of each module.

7 Each **Build** is a version of the software which offers a subset of its behaviour. Typically early builds contain only infrastructure software and little application functionality. Each build acts as a test harness for subsequent code.

8 The **Installed Software** is the final build, configured and installed in its operational environment.

Section 3 describes each of these deliverables in more detail.

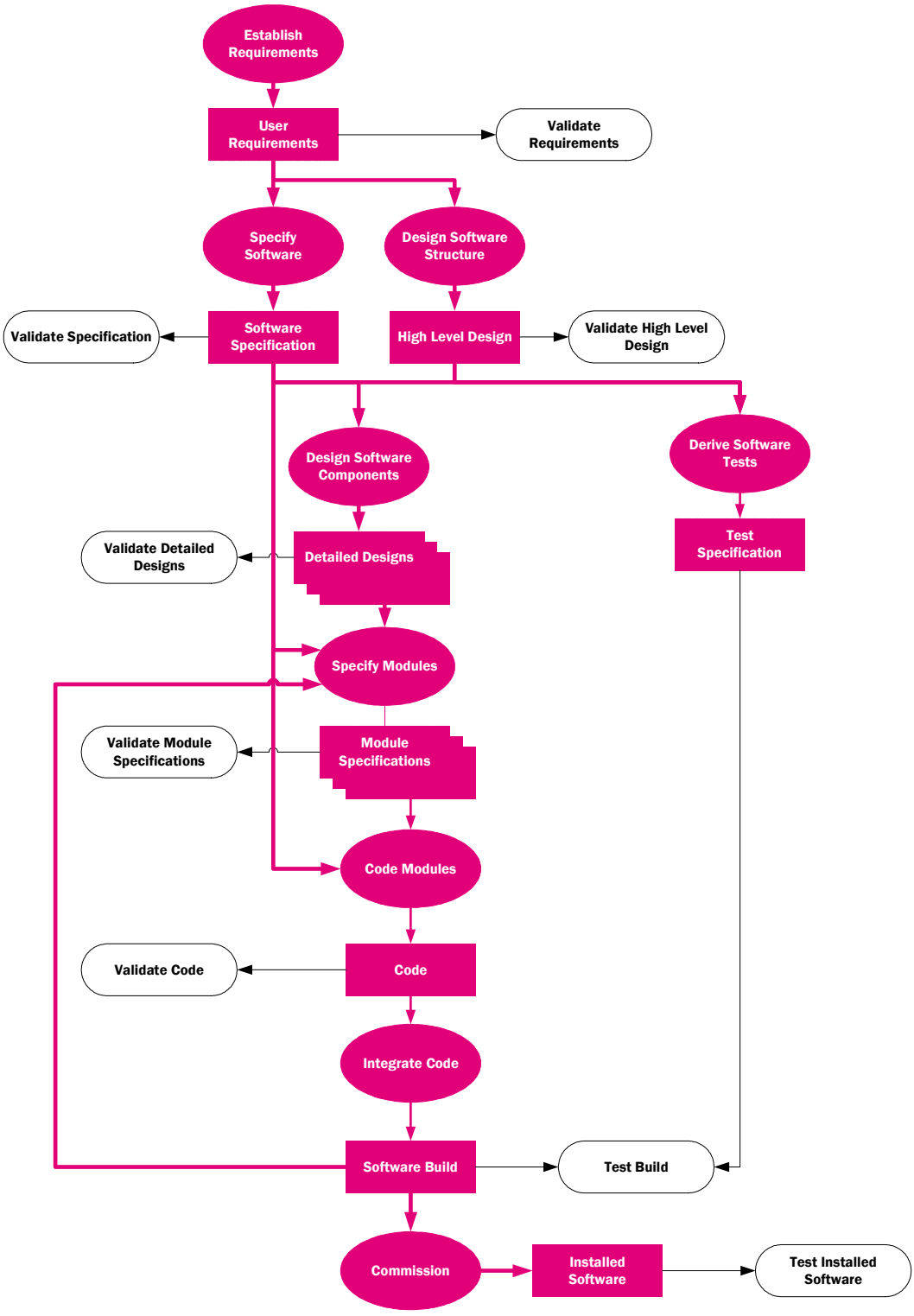

Figure 1 Core Process

This diagram is simplified by omitting most of the parallelism and iteration.

Figure 2 Key to Process Diagram

2.2 Process Characteristics

2.2.1 Risk Driven

We choose the set of activities and the order we do activities to minimise the risk of late problems. We therefore do the most risky activities first. If, for example, we are uncertain about the feasibility of meeting some requirement, we will do design trailblazing to establish a feasible design ahead of completing the requirements or specification. We also choose how much specification and design to do on the basis of risk: if an area is straightforward, we may go straight from requirements to code, while in a difficult area we will write very detailed and formal specifications.

2.2.2 Confidence building

The Correctness by Construction process provides evidence, throughout the process, about the correctness of the software being built. This evidence builds confidence that there will be no late-breaking serious faults. It also supports evaluation and certification of security-critical software, for

example against the Common Criteria, and of safety-critical software, for example against DEF STAN 00-55.

2.2.3 Parallel

Although the figure appears to describe a largely sequential process, we actually use a lot of parallelism to reduce timescales. There are three ways we can achieve this:

1 Where two different kinds of activity are independent, we do them in parallel. For example, the high level design is based largely on non-functional requirements and does not depend on details of the functional specification, so it can be done in parallel with the software specification.

2 Where the system can be partitioned into different areas, these can be developed in parallel. They may be at different stages of development at the same time, or progress through the same stages at the same time.

3 Incremental builds allow us to carry out testing of one build in parallel with coding of the subsequent build.

2.2.4 Iterative

Whenever we find a fault, we iterate back to the point at which the fault was introduced and rework all subsequent deliverables. (Obviously we do this in batches, not for each individual fault.) This ensures that all deliverables are kept consistent at all baselines.

2.2.5 Rigorous

At each stage, we use descriptions that are as formal as possible. This has two benefits. First, formal descriptions are more precise than informal ones, and therefore they force us to understand issues and questions before we actually produce the code. Second, there are more powerful verification methods for formal descriptions than there are for informal ones, so we have more confidence in the correctness of each step. In particular, formal methods allow some degree of automated checking of the deliverables and of the relationships between them.

2.2.6 Early validation

The aim of correctness by construction is to prevent faults and to eliminate as early as possible any faults that are introduced. Therefore each deliverable is validated as rigorously as possible. Wherever possible we use formal notations and automated tools to validate specifications and designs before any faults get through to code.

2.2.7 Efficient

There are two reasons why Correctness by Construction is an efficient process. The first is that it minimises late rework. Because faults are detected and removed as early as possible, few faults survive to the late stages of the project. The second is that it minimises duplication and repetition of work. The deliverables all describe different aspects of the software and there is little overlap between them. This contrasts with methods where each deliverable is essentially an expansion of the previous one.

2.2.8 Measured

We keep metrics on size, productivity and defect rates across the process.

2.2.9 Improved through root cause analysis

We do root cause analysis of significant faults and continuously improve the process.

2.2.10 Flexible

Correctness by construction is not a single, rigid process. Rather it is a framework and set of principles. For any particular project we tailor it based on the nature and criticality of the project. Projects may differ in many aspects:

1 Level of rigour
Some projects require fully formal proofs of correspondence between formal specifications; others may not justify any formality at all.

2 Techniques/notations at each stage
Different kinds of software require different notations. For example embedded systems need a very different style of specification from database applications.

3 Subsets of activities
Some projects may omit some of the activities, or add extra activities.

4 Content of design
The amount of detail in the design will depend on the size and complexity of the system.

5 Formality of evaluation
The amount of evidence that is collected and the rigour with which the evidence is controlled can be adapted according to how rigorously the software is to be evaluated. The process is capable of developing to software to the highest levels of safety (for example safety integrity level 4 as defined by UK MoD DEF STAN 00-56) and security (for example Common Criteria assurance level EAL 7).

3 Process Steps

This section describes the production and validation of each of the process deliverables.

3.1 Requirements

The User Requirements define:

1 The overall objectives of the system.

2 The system context: the people and other systems that interact with it.

3 Relevant facts about the application domain.

4 Functions to be provided by the system and scenarios showing how they achieve the overall objectives.

5 Non-functional characteristics such as capacity, throughput, reliability, safety and security.

We establish and describe the requirements using Praxis' REVEAL® method. They may be captured in a document or in a requirements tool such as DOORS or Requisite Pro.

We pay particular attention to the changeability of requirements. We identify those requirements and assumptions that are relatively stable, and those that are more likely to change. This allows us to design the system to cope with the likely changes that will occur during its development and use.

The requirements for secure systems include the security target. This will be stated in English language. For high levels of assurance, we also write a Formal Security Policy Model. This formalises the technical aspects of the security target. This has two benefits:

1 It makes the security target absolutely precise.

2 It allows more rigorous validation of subsequent deliverables.

The user requirements are validated by review. The review includes the users of the system and also developers and testers, who ensure that the requirements are feasible and testable.

3.2 Specification

The specification is a complete black-box description of the behaviour of the software. It describes several aspects:

1 Functionality
We specify the functionality by writing an abstract description of the system state and a description of the effect of each operation in terms of inputs, outputs and state changes. We always give a

complete description including both normal and error outcomes. We write this description in a formal notation such as Z [4], although we also use notations such as UML class diagrams to describe the system state and, in some cases, state diagrams to describe behaviour.

2 Concurrency
Some systems have a high degree of concurrency visible to the users and there may be rules about what concurrent behaviours are allowed. If so we describe these rules, using a formal language such as CSP [5].

3 User Interface
We describe in detail the required look and feel of the user interface.

4 System Interfaces
We produce an interface control document defining the interface with each connected system.

We validate the software specification by:

1 Review
This is a manual check that the specification is
   a self consistent (both within each part, and that the abstract specifications are consistent with the user and system interfaces);
   b correct with respect to the user requirements;
   c complete;
   d implementable.

2 Prototyping
We build a prototype of the user interface and evaluate it with suitable user representatives. We may also build prototypes of critical functionality to validate the abstract specification.

3 Formal Analysis
When we have a formal specification we can carry out proofs to show that it is self consistent and has some completeness properties. In critical secure applications we can also prove correspondence between the specification and the formal security policy model.

3.3 High Level Design

The high level design is a top level description of the system's internal structure and an explanation of how the components worked together. There are several different descriptions, looking at the structure from different points of view. In a secure system the descriptions typically include:

1 distribution of functionality over machines and processes;

2 database structure and protection mechanisms;

3 mechanisms for transactions and communications.

We use different notations for the different aspects of the design. Not all aspects have formal notations. Where possible we do use formal notations: in particular we use CSP to define any complex process structure.

The high level design is primarily derived from the non-functional requirements and can be developed in parallel with the software specification.

We validate the high level design by:

1 Review, to ensure that it

   a is self consistent;

   b satisfies the requirements;

   c is implementable.

2 Automated analysis
   If we have a formal design in CSP we use automated tools such as model checkers to validate that the design has desired properties such as freedom from deadlock.

3.4 Detailed Design

The detailed design serves two purposes

1 defining the set of software modules and processes and allocating the functionality across them;

2 providing more detail of particular modules wherever that is necessary.

3.4.1 Module Structure

The module structure describes the software architecture and how functionality described in the specification and high-level design is allocated to each module. We recognize that the structure of the implementation may differ from that of the specification, for example where an atomic transaction in the specification is distributed over many processes or machines in the implementation.

We use an information-flow centric design approach, called INFORMED, to drive and evaluate the module structure. We developed INFORMED within Praxis Critical Systems to support the design of high integrity software. It leads to a software architecture that exhibits low coupling and high cohesion. This, in turn, benefits the later maintainability of the system in the face of subsequent changes. For secure systems, we categorize system state and operations according to their impact on security. We aim for an architecture that minimizes and isolates security-critical functions, so reducing the cost and effort of the (possibly more rigorous) verification of those units. We use a similar approach for safety-critical software.

For embedded and real-time systems, we also consider throughput, timing and scheduling issues at this stage. Systems with particularly stringent hard real-time requirements, for example, might constrain the style of implementation architecture that can be employed.

3.4.2 Low-level details

We deliberately do not write a detailed design of every aspect of the system. Often the software specification and module structure together give enough information to create software directly. We do not duplicate information that is already in the specification or HLD. However we may provide more detailed designs for, for example:

1 database table structures;

2 complex areas of functionality: these arise where there is a big difference between the implementation structure and the conceptual structure in the software specification, or where the software specification has omitted details of some complex processing;

3 user interface code;

4 low-level device handling;

5 rules for mapping specification or design constructs into code: for example, rules for translating CSP into Ada tasking constructs or Z types into implementation structures.

The detailed designs use different notations for different aspects. We use formal specifications of low-level modules to clarify complex functionality.

The detailed design is validated by

1 review, to ensure that it is self consistent, efficient and satisfies the specification and the high level design;

2 formal analysis
A formal low-level specification can be proved correct with respect to a higher level specification.

3.5 Test Specifications

We derive test specifications primarily from the software specification, together with the requirements and the high-level design. We use boundary value analysis to generate tests which cover the specification. We then supplement these with tests for behaviour which is introduced by the design but is not visible in the specification. In addition we generate tests for non-functional requirements directly from the requirements document.

3.6 Module Specifications

For each module identified in the module structure, we may construct a more detailed specification for its implementation. We may code directly from the system specification if that specification is already suitably detailed and the "gap" between the specification and code is sufficiently narrow.

The module specification serves as a contract between the system specification or a detailed design and the code itself. We favour languages that have a well-defined semantics and directly support design-by-contract. Depending on the application domain, the module specification may be expressed in a model-based notation (such as UML class specifications), an executable specification (such as Statecharts or control-law specifications), or a design-by-contract programming language (such as Eiffel or SPARK).

We validate the module specifications by review, using tools as far as possible, to check for internal consistency, validity with respect to the system specification and so on.

3.7 Code

For coding, we use languages and tools that are most appropriate for the task at hand. Validation requirements play a large role in this choice—languages must be amenable to verification and analysis so that the required evidence of fitness-for-purpose can be generated effectively. We also recognize that no one language is most suitable for all modules—in a secure system, for example, we might use different languages for the security kernel, the system's infrastructure, and the user-interface.

Code is derived from the system specification, module specifications, and low-level designs. We use automatic code generation where domain-specific tools (such as GUI-builders or control system design packages) are mature.

We validate code using both static and dynamic techniques. We use static verification tools are far as possible, since these prevent and detect defects earlier in the life-cycle than testing would allow. Such tools range from simple style and subset-checking up to fully formal program verification systems. We always perform manual code-review, although this is only ever performed after application of static analysis, and we tune the review process to account for the classes of defect that the tools can eliminate.

3.8 Building

We build the system top down and incrementally, with a formal build every few weeks. The first build consists of the system framework, such as the top-level processes on each machine and the connections to the external devices. It does not, typically, contain much application functionality. Each build acts as a test harness for later code.

We test each build by running a subset of the system tests. We use automated regression testing to ensure that all previous tests are still passed. We measure code coverage as we carry out the tests. When we find gaps in coverage, we do one of 3 things:

1 Usually the gap is caused by code which is there to implement some aspect of the design not visible at the specification level. In that case we add suitable tests.

2 Sometimes the gap reflects code which is not in fact necessary, and the code is removed.

3 Sometimes the code cannot be reached by normal operation of the system, but is still necessary – for example defensive code. In that case, and only then, we write unit tests at the module level.

Apart from this last case, we do no formal unit testing. Unit testing is costly and ineffective at finding errors in comparison with proof and static analysis [3].

3.9 Commissioning

The commissioning process largely depends on the criticality of the system being delivered and its operational environment. At the least, we use a documented process for the labelling and delivery of software builds to the customer. A build is accompanied by a "release certificate" that summarizes the status and composition of that particular build. For some applications, we also issue a safety certificate and a warranty.

We supply a Commissioning Guide to the customer, which details the installation of the software onto the target environment. This may also contain details of how the system's hardware is constructed, and an inventory of the required components.

For secure systems, we go further. Software may be delivered in tamper-evident bags, for example, according to the customer's (and regulator's) requirements. Such systems may be commissioned in a physically secure environment, and commissioning may be witnessed by us, the customer, independent auditors, evaluators, regulators and so on.

4 Generic Activities

4.1 Process Planning

We write a technical plan for each project. This describes what parts of the process we will use, what techniques we will use at each stage and what validation activities we will carry out. Thus the process is tailored for each particular application.

4.2 Staff Competence and Training

We ensure that all staff working on a project are competent in the relevant areas. We use in-house or bought-in training to maintain our skill levels.

4.3 Tracing

We maintain tracing information showing how each description is related to its successor and predecessor. For example we record how each requirement is satisfied in the specification. We use automated tools to check that all requirements are completely traced through to code, and that all code is ultimately traceable back to the requirements.

4.4 Fault management

Fault management is a key part of Correctness by Construction, since the whole aim is to remove faults as early as possible. Each deliverable is subject to fault management as soon as it has been baselined. Faults are identified by the validation activities and also by use of deliverables in subsequent stages – for example, a coder may find a fault in the specification.

When a fault is identified, we do two things:

1 Identify and fix all the deliverables affected by the fault. These may include deliverables earlier than that in which the fault was first identified, as well as deliverables derived from it.

2 Do root cause analysis to determine why the fault was introduced, and if possible change the process to avoid faults of this sort appearing in future.

4.5 Change management

The key to change management is impact analysis. We find that the rigorous specification and design information makes impact analysis highly effective. For each change we are able to give a detailed assessment of the effect on each deliverable.

Change management is also helped by design for change, starting with the changeability requirements described in section 3.1.

4.6 Configuration management

All deliverables including documents as well as code are under formal tool-supported configuration management. This enables us to identify versions of individual items and baseline configurations of consistent sets of documents and code.

4.7 Team organisation

For critical projects we do all formal testing using a team independent of the implementers.

4.8 Metrics collection

We collect metrics on effort, sizes of deliverables, numbers of faults and for each fault its point of introduction and point of detection.

5 Examples of Process Use

This section presents metrics for five projects that have used instances of the Correctness-by-Construction approach. These projects differ in size, application domain and complexity, although all are classed as "high integrity"—three of the projects have critical safety-relation functions, while the other two have significant security requirements.

The following paragraphs give a brief description of each project. The projects are identified by name where, given clients' confidentiality, we are able to do so at the time of writing.

5.1 CDIS

CDIS is a real-time air traffic information system. It has stringent performance and availability requirements and has proved very reliable in over 11 years of use at the London Terminal Control Centre. The methods used in developing CDIS have been described in an article [9] and there has been an independent assessment of the project [10].

5.2 SHOLIS

The Ship/Helicopter Operating Limits Information System aids the safe operating of helicopter operations on naval vessels. It is essentially an information system, giving advice on the safety of helicopter operations given a particular operating scenario and environmental conditions such as the incident wind vector and the roll and pitch of the ship. The system's primary safety function is to raise audible and visible alarms when environmental conditions step outside of pre-defined operating limits.

SHOLIS was the first project to carry out a full SIL 4 development under the UK MoD's Def Stan 00-55. Further information on SHOLIS can be found in [3].

5.3 The MULTOS CA

The MULTOS CA is the "root" certification authority for the MULTOS smartcard system. The CA produces digital certificates that are used in the manufacturing of MULTOS smartcards, and also certificates that allow trusted applications to be loaded onto a card in the field. The system has demanding throughput and availability requirements, and so is both distributed and fault-tolerant.

The CA was developed as far as was practicable in line with the UK ITSEC scheme at evaluation level E6—roughly equivalent to Common Criteria EAL7. Further information on the development of the CA can be found in [1].

5.4 Project A

This project is a military stores management system. It enforces a small number of safety functions, and was developed in line with the UK's Def Stan 00-55[8] standard at SIL 3. This project is embedded, and combines a simple user-interface with complex hard real-time requirements.

5.5 Project B

This project is the core of a biometric access-control system. It has been developed using Correctness-by-Construction to meet or exceed the requirements of the Common Criteria at evaluation/assurance level EAL5.

5.6 Metrics

Table 1 presents key metrics for each of the above projects. The first column identifies each project. The second identifies the year in which the system was first commissioned—the projects are presented in chronological order. Column three shows the size of the delivered system in physical lines of code. This is always executable lines and declarations, but does not include comments, blanks lines, or "annotations" used for design-by-contract. The fourth column presents productivity—this is the lines of code divided by the total project effort for **all** project phases from project start up to the completion of commissioning. The final column reports defect rate in defects per thousand lines of code.

Project	**Year**	**Size (loc)**	**Productivity (loc per day)**	**Defects (per kloc)**
CDIS	1992	197,000	12.7	0.75
SHOLIS	1997	27,000	7.0	0.22 (note 1)
MULTOS CA	1999	100,000	28.0	0.04 (note 2)
A	2001	39,000	11	0.05 (note 3)
B	2003	10,000	38.0	not yet known (note 4)

Table 1: Correctness-by-Construction project metrics

Notes:

1. 0 defects during acceptance test and sea-trial. 6 defects subsequently discovered and corrected in first 3 years of in-service use.

2. 4 defects reported and corrected during 1-year warranty period following commissioning.

3. 2 defects in 2 years following delivery. However, the system is not yet rolled out for operational service.

4. This project has been undergoing independent evaluation for some months. No defects have been detected so far but the final results will not be available until February 2004.

A SPARK

The SPADE Ada Kernel (SPARK) is a language designed for the construction of high-integrity systems. The executable part of the language is a subset of Ada95, but the language requires additional annotations that make it possible to carry out data and information flow analysis[6], and to prove properties of code, such as partial correctness and freedom from exceptions.

The design goals of SPARK are as follows:

Logical soundness: there should be no ambiguities in the language;

Simplicity of formal description: it should be possible to describe the whole language in a relatively simple way;

Expressive power: the language should be rich enough to construct real systems;

Security: it should be possible to determine statically whether a program conforms to the language rules;

Verifiability: formal verification should be theoretically possible and tractable for industrial-sized systems.

The annotations in SPARK appear as comments (and so are ignored by a compiler), but are processed by the Examiner tool. These largely concern strengthening the "contract" between the specification and the body of a unit (for instance specifying the information flow between referenced and updated variables.) The presence of the annotations also enables the language rules to be checked efficiently, which is crucial if the language is to be used in large, real-world applications.

SPARK actually has its roots in the security community. Research in the 1970's[7] into information flow in programs resulted in SPADE Pascal and, eventually, SPARK. SPARK is widely used in safety-critical systems, but we believe it is also well-suited to the development of secure systems. SPARK offers static protection from several of the most common implementation flaws that plague secure systems:

- **Buffer overflows**. Proof of the absence of predefined exceptions (for such things as buffer overflows) offer strong *static* protection from a large class of security flaw. Such things are an anathema to the safety-critical community, yet remain a common form of attack against networked computer systems. The process of attempting such proofs also yields interesting results: a proof which doesn't "come out" easily often is indicative of a bug, and the proof *forces* an engineer to read, think about, and understand their programs in depth, which can only be a good thing.

- **Run-Time Library Defects.** SPARK can be compiled with no supporting run-time library, implying that an application can be delivered with no COTS component. At the highest assurance levels, this may be of significant benefit, where evaluation of such components remains problematic.

- **Timing and memory attacks.** SPARK is amenable to the *static* analysis of timing and memory usage. This problem is known to the real-time community, where analysis of worst-case execution

time is often required. In the development of secure systems, it may be possible to use such technology to ensure that programs exhibit as little *variation* in timing behaviour as possible, as a route to protect against timing analysis attacks.

- **Input Data Validation.** The SPARK verification system is conservative, and does not trust data coming from the external environment. Formally speaking, the verification condition generator does not automatically add hypotheses regarding input data, so that subsequent proofs (e.g. for a range check where such an input is used) cannot be discharged until the validity of that input has been explicitly checked. In short, SPARK forces the programmer to validate input data (or at least provides a very strong reminder to do so!)

Additionally, SPARK provides additional forms of verification, such as:

- Program-wide, complete data- and information-flow analysis. These analyses make it impossible for a SPARK program to contain a dataflow error (e.g. the use of an uninitialized variable)—a common implementation error that can be the cause of subtle (and possibly covert) security flaws.

- Proof of correctness of SPARK programs is achievable, and so allows a program to be shown to correspond with some suitable formal specification. This allows for formality in the design and specification of a system to be extended through its implementation and can meet the requirements of the CC scheme at the highest evaluation levels.

More information about SPARK can be found at www.sparkada.com

Document Control and References

Changes history

Issue 0.1 (12th January 2004): First draft for internal inspection.

Issue 1.0 (12th January 2004): First external issue following inspection.

Issue 1.1 (13th January 2004): Management summary added and minor corrections made.

Changes forecast

May subsequently be developed further.

Document references

1 *Correctness by Construction: Developing a Commercial Secure System,* Anthony Hall and Roderick Chapman, *IEEE Software* Jan/Feb 2002, pp18-25.

2 *Will it Work?*, Jonathan Hammond, Rosamund Rawlings and Anthony Hall, in *Proceedings of RE'01, 5th IEEE International Symposium on Requirements Engineering*, August 2001

3 *Is Proof More Cost-Effective Than Testing?*, Steve King, Jonathan Hammond, Rod Chapman and Andy Pryor, IEEE Transactions on Software Engineering, Vol 26 No 8, pp675–686, August 2000

4 *The Z Notation: A Reference Manual*, J. M. Spivey, 2nd Edition. Prentice-Hall, 1992.

5 *Communicating Sequential Processes*, C. A. R. Hoare, Prentice Hall, 1985.

6 Bergeretti, J-F., and Carré, B. A., *Information-Flow and Data-Flow Analysis of While Programs*. ACM Transactions on Programming Languages and Systems, Vol. 7, No. 1, January 1985. p.p. 37–61.

7 Denning, D. E., and Denning, P. J. *Certification of Programs for Secure Information Flow*. CACM Vol. 20, No. 7. July 1977.

8 United Kingdom Ministry of Defence, *The Procurement of Safety Critical Software in Defence Equipment.* DEF STAN 00-55.

9 *Using Formal Methods to Develop an ATC Information System*, Anthony Hall, IEEE Software, March 1996, pp 66-76.

10 *Investigating the Influence of Formal Methods*, Shari Lawrence Pfleeger and Les Hatton, IEEE Computer, February 1997, pp 33-43.