Correctness by Construction: Developing a Commercial Secure System

Anthony Hall and Roderick Chapman

Publication notes

Published in IEEE Software, Jan/Feb 2002, pp18-25.

Copyright © 2002 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

© 2002 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Correctness by Construction: Developing a Commercial Secure System

Anthony Hall and Roderick Chapman, *Praxis Critical Systems*

Praxis Critical Systems recently developed a secure Certification Authority for smart cards. The CA had to satisfy demanding performance and usability requirements while meeting stringent security constraints. The authors show how you can use techniques such as formal specification and static analysis in a realistic commercial development.

When you buy a car, you expect it to work properly. You expect the manufacturer to build the car so that it's safe to use, travels at the advertised speed, and can be controlled by anyone with normal driving experience. When you buy a piece of software, you would like to have the same expectation that it will behave as advertised. Unfortunately, conventional software construction methods do not provide this sort of confidence: software often behaves in completely unexpected ways. If the software in question is security- or safety-critical, this uncertainty is unacceptable. We must build software that is correct by construction, not software whose behavior is uncertain until after delivery.

Correctness by construction is possible and practical. It demands a development process that builds correctness into every step. It demands rigorous requirements definition, precise system-behavior specification, solid and verifiable design, and code whose behavior is precisely understood. It demands defect removal and prevention at every step. What it does not demand is massive spending beyond the bounds of commercial sense. On the contrary, the attention to correctness at early stages pays off in reduced rework costs. When we build software in this way, we give a warranty that it will behave as specified, and we don't lose any sleep over the cost of honoring this warranty.

This article describes how we applied this philosophy to the development of a commercial secure system. The system had to meet normal commercial requirements for throughput, usability, and cost as well as stringent security requirements. We used a systematic process from requirements elicitation through formal specification, user interface prototyping, rigorous design, and coding in Spark, to ensure these objectives' achievement. System validation included tool-assisted checking of a formal process design, top-down testing, system testing with coverage analysis, and static code analysis. The system uses commercial off-the-shelf hardware and software but places no reliance on COTS correctness for critical security properties. We show how a process that achieves normal commercial productivity can deliver a highly reliable system that meets all its throughput and usability goals.

0740-7459/02/$17.00 © 2002 IEEE

List of Abbreviations

CA	Certification Authority
COTS	commercial off-the-shelf
FSPM	Formal Security Policy Model
FTLS	formal top-level specification
HLD	high-level design
ITSEC	Information Technology Security Evaluation Criteria
UIS	user interface specification
UR	user requirements

Background

Praxis Critical Systems recently developed the Certification Authority for the Multos smart card scheme on behalf of Mondex International (MXI).[1] (See the "List of Abbreviations" and "Useful URLs" sidebars for more information.) The CA produces the necessary information to enable cards and signs the certificates that permit application loading and deletion from Multos cards.

Obviously, such a system has stringent security constraints. It must simultaneously satisfy commercial requirements for high throughput and good usability by its operators. The combination of security and throughput requirements dictated a distributed system with several processors. Furthermore, to meet the development budget and timescale, we could not build the system from scratch, requiring use of COTS hardware and infrastructure software.

MXI was keen for a predictable development process, with no surprises and minimum risk. They also wanted to develop according to the UK Information Technology Security Evaluation Criteria,[2] one of the forerunners of the Common Criteria.[3] The CA supports smart cards that are certified to the highest ITSEC level, E6. This requires a stringent development process including the use of formal methods at early stages. Previous experience had shown that, when properly applied, the E6 process forced the customer and supplier to explicitly and unambiguously understand system requirements, which avoided unpleasant surprises in late testing. We therefore developed the CA to the standards of E6.

The development approach

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

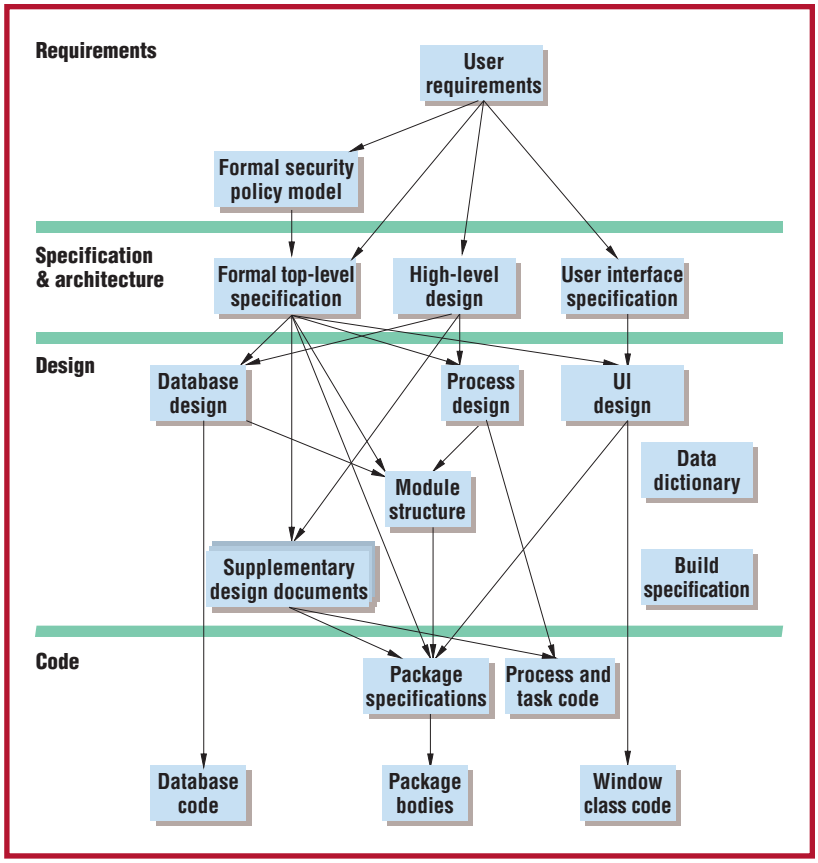


Figure 1. Development deliverables grouped into the main process steps.

At each stage, we used descriptions that were as formal as possible. This had two benefits. Formal descriptions are more precise than informal ones, which forced us to understand issues and questions before we actually got to the code. Additionally, more powerful verification methods for formal descriptions exist than methods for informal ones, so we had more confidence in each step's correctness.

Figure 1 shows the overall set of deliverables from the development process, grouped into the main process steps.

Requirements

Before we thought about the CA system,

Useful URLs

Spark	www.sparkada.com
Mondex	www.mondex.com
Multos	www.multos.com
ITSEC	www.cesg.gov.uk
Common Criteria	www.commoncriteria.org
Formal methods and Z	www.afm.sbu.ac.uk

we had to understand the business environment in which it would operate. We used our well-tried requirements-engineering method, Reveal,[4] to define the CA's environment and business objectives and to translate these into requirements for the system.

We wrote the user requirements (UR) document in English with context diagrams,[5] class diagrams, and structured operation definitions,[6] providing a first step toward formalizing the description. We labeled every requirement so that we could trace it, and we traced every requirement to its source. We traced each security requirement to the corresponding threats. We carried out this tracing through all the development documents down to the code and tests. We validated user requirements through client review and manual verification for consistency between different notations.

The highest levels of ITSEC and the Common Criteria require a Formal Security Policy Model (FSPM). The user requirements included an informal security policy that identified assets, threats, and countermeasures. This contained 28 technical items. Of these, we formalized the 23 items related to viewing the system as a black box rather than the five that dealt with implementation details.

Specification and architecture

This phase covered two activities, carried out in parallel:

- detailed system behavior specification and
- high-level design (HLD).

The system specification comprises two closely related documents: the user interface specification and the formal top-level specification. Together, these completely define the CA's black-box behavior.

The UIS defines the system's look and feel. We developed it by building a user interface prototype, which we validated with the CA operational staff.

The FTLS defines the functionality behind the user interface. We derived it from the functions identified in the UR, the constraints in the FSPM, and the user interface prototype's results. We used a typechecker to verify that the FTLS was well formed, and we validated it by checking against the user requirements and FSPM and by client review.

The HLD contained descriptions of the system's internal structure and explanations of how the components worked together. Several different descriptions looked at the structure in varying ways, such as in terms of

- distribution of functionality over machines and processes,
- database structure and protection mechanisms, and
- mechanisms for transactions and communications.

The HLD aimed mainly to ensure satisfaction of security and throughput requirements. One of the most important parts was achieving security using inherently insecure COTS components, such as a commercial database. We did this on the basis of our experience using COTS in safety-critical systems. Specifically, we did not rely on COTS for confidentiality or integrity. We achieved these by

- hardware separation;
- confidential information encryption;
- message authentication codes, where data integrity was needed; and
- individual processing of security-critical functions to avoid reliance on separation between processes.

Detailed design

The detailed design defined the set of software modules and processes and allocated the functionality across them. It also, when necessary, provided more detail for particular modules.

We deliberately did not write a detailed design of every system aspect. Often, the FTLS and module structure gave enough information to create software directly. However, a few cases existed, where the implementer required much more information than the FTLS had. For example, we used Z[7]—a mathematical language supported by English descriptions—to specify the module that manages cryptographic keys and their

verification on system startup. This specification helped us resolve many difficult issues before coding.

We used different notations for the design documents, according to their subject matter, always following the rule of maximum practical formality. The most novel part was the process design.

Code

Coding the CA posed some interesting challenges. For example, the CA's expected life span is some 20 years. With this in mind, we viewed development technologies that were particularly fashionable at the time with some skepticism.

Additionally, although most development favors the reuse of COTS components, we tried to avoid these in the CA's development as far as was practical. For instance, we implemented a remote procedure call mechanism entirely from scratch, rather than relying on commercial middleware such as DCOM or an implementation of Corba.

We aimed for *five nines* availability (that is, 99.999 percent) in security-critical parts. Housed in a tamperproof environment, the system cannot be rebooted without some effort, so we spent considerable effort in ensuring the system's stability. We estimated actual maintenance of the system's tamperproof parts (for example, installation of new software builds) to be possible only once every six months.

Other challenges included the client's requirement of a commercial operating system (in this case, Windows NT 4) and the ITSEC (and more recently the Common Criteria) requirements for languages that "unambiguously define the meaning of all statements" used in a system's implementation.

We soon realized that no single implementation language could do the job. Our experience with safety-critical system development suggested that implementation in Ada95 would suit the system's critical parts. However, Ada95 was clearly not appropriate at the time for the GUI's development. Ultimately, we settled on a "right tools for the job" mix of languages. So, we implemented the system's security-enforcing kernel in Spark Ada[8]—an annotated subset of Ada95 widely used in safety-critical systems—whose properties make it suitable for secure system development. (See the "Spark and the Development of Secure Systems" sidebar for more information.) We implemented the system's infrastructure (for example, remote procedure call mechanisms and concurrency) in Ada95. The system's architecture carefully avoids any security-related functionality in the GUI, so we implemented this in C++, using Microsoft's Foundation Classes. We used some small parts, such as device drivers for cryptographic hardware, and one standard cryptographic algorithm as is. We reviewed the C source code for these units by hand.

We identified some of the system's technical aspects early on as carrying some risk. These included the implementation of a concurrent Ada program as a Win32 service, use of the Win32 and Open Database Connectivity APIs from Ada, and linking the C++ GUI with the underlying Ada application software. We attacked these risks using "trailblazing" activities such as implementing small demonstrator applications.

For all system parts, we enforced rigorous coding standards. We reviewed all the code against these standards and relevant source documents, such as the FTLS and UIS. We also used automatic static-analysis tools where possible, such as the Spark Examiner for Ada, and BoundsChecker and PC-Lint for C++.

Verification and validation

Correctness by construction does not claim zero defects: we do not believe that this is achievable in any engineering artifact. We do, however, have zero tolerance of defects. We try to find them as early as possible, and then we eliminate them. Furthermore, we collect data on defect introduction and removal and try to improve the process to reduce introduction of defects and to speed defect discovery.

The first line of attack is review. We review all deliverables to check

- correctness with respect to earlier deliverables,
- conformance to standards, and
- internal consistency.

Wherever possible, we carry out automated verification and validation on deliverables. As you'll see in the next section, we were able to do some automated checks on

Although most development favors the reuse of COTS components, we tried to avoid these in the CA's development as far as was practical.

Spark and the Development of Secure Systems

The Spade Ada Kernel (Spark) is a language designed for constructing high-integrity systems. The language's executable part is a subset of Ada95, but the language requires additional annotations to let it carry out data- and information-flow analysis[1] and to prove properties of code, such as partial correctness and freedom from exceptions.

These are Spark's design goals:

- *Logical soundness.* The language should have no ambiguities.
- *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.

Spark annotations appear as comments (and so are ignored by a compiler) but are processed by the Examiner tool. These largely concern strengthening the "contract" between a unit's specification and body (for instance, specifying the information flow between referenced and updated variables). The annotations also enable efficient checking of language rules, which is crucial for using the language in large, real-world applications.

Spark has its roots in the security community. Research in the 1970s into information flow in programs[2] resulted in Spade Pascal and, eventually, Spark. Spark is widely used in safety-critical systems, but we believe it is also well suited for developing secure systems.

Particularly, it offers programwide, complete data- and information-flow analysis. These analyses make it impossible for a Spark program to contain a dataflow error (for example, the use of an uninitialized variable), a common implementation error that can cause subtle (and possibly covert) security flaws.

You can also achieve proof of correctness of Spark programs, which lets you show that a program corresponds with some suitable formal specification. This allows formality in a systems' design and specification to be extended throughout its implementation.

Proof of the absence of predefined exceptions (for such things as buffer overflows) offers strong static protection from a large class of security flaw. Such things are anathema to the safety-critical community yet remain a common form of attack against networked computer systems. Attempting such proofs also yields interesting results. A proof that doesn't come out easily often indicates a bug, and the proof forces engineers to read, think about, and understand their programs in depth. Experience on other projects suggests that proof is a highly cost-effective verification technique.[3]

You can compile Spark without a supporting runtime library, which implies that you can deliver an application without a commercial off-the-shelf component. This might offer significant benefits at the highest assurance levels, where evaluation of such components remains problematic.

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. When developing secure systems, you might be able to use such technology to ensure that programs exhibit as little variation in timing behavior as possible, as a route to protect against timing-analysis attacks. You can access more information about Spark at www.sparkada.com.

References

1. J-F. Bergeretti and B.A. Carré, "Information-Flow and Data-Flow Analysis of While Programs," *ACM Trans. Programming Languages and Systems*, vol. 7, no. 1, Jan. 1985, pp. 37–61.
2. D.E. Denning and P.J. Denning, "Certification of Programs for Secure Information Flow," *Comm. ACM*, vol. 20, no. 7, July 1977, pp. 504–513.
3. S. King et al., "Is Proof More Cost-Effective Than Testing?" *IEEE Trans. Software Eng.*, vol. 26, no. 8, Aug. 2000, pp. 675–686.

formal specifications and designs, which helped to remove defects early in the process.

Our main verification and validation method is, of course, testing. Traditionally, critical-systems testing is very expensive. A reason for this is that testing occurs several times: we test individual units, integrate them and test the integration, and then test the system as a whole. Our experience with previous safety-critical projects[9] suggests that this approach is inefficient and particularly that unit testing is ineffective and expensive. Unit testing is ineffective because most errors are interface errors, not internal errors in units. It is expensive because we must build test harnesses to test units in isolation.

We adopted a more efficient and effective approach. We incrementally built the system from the top down. Each build was a real, if tiny, system, and we could exercise all its functions in a real system environment. This reduced the integration risk. We derived the tests directly from the system specification. We ran the tests using Rational's Visual Test, so that all tests were completely automated. Furthermore, we instrumented the code using IPL's AdaTest so that we measured the statement and branch coverage we were achieving by the system tests. We devised extra design-based test scenarios only where the system tests failed to cover parts of the code.

Formal methods

We used Z to express the FSPM. We

based our approach on the Communications-Electronics Security Group's Manual "F"[10] but simplified the method.

The informal security policy contained four different kinds of clause, each giving rise to a different kind of predicate in the FSPM:

- Two clauses constrained the system's overall state (each became a state invariant in the formal model).
- Eight clauses required the CA to perform some function (for example, authentication).
- Sixteen clauses were constraints applicable to every operation (for example, that only authorized users could perform them).
- One clause was an information separation clause.

Information separation is harder to express in Z than other properties, and other formal languages such as CSP can express it more directly. However, we found the other 24 clauses straightforward to express in Z, so Z proved a good choice of language overall.

Because we wrote the FSPM in Z, we could check some aspects of its internal consistency using a typechecker. We reviewed it for correctness with respect to the informal security policy. We did not carry out any proofs of correctness; although, in other projects, we found these to be effective in finding errors.[9]

Formal top-level specification

The FTLS is a fairly conventional Z specification. However, it contains some special features to allow checking against the FSPM. In conventional Z, one or two schemas express an operation. In the FTLS, we used numerous schemas to capture each operation's different security-relevant aspects.

We used separate schemas to define each operation's inputs, displayed information, and outputs. This let us trace clearly to FSPM restrictions on what is displayed and how outputs are protected. We used separate schemas to define when an operation was available or valid. This let us distinguish both errors that are prevented by the user interface and those that are checked once we confirm the operation and thus cause error messages to be displayed. We also modeled errors in detail, to satisfy the requirement of reporting all errors.

Process design

We modeled the process structure in the CSP language. We mapped sets of Z operations in the FTLS to CSP actions. We also introduced actions to represent interprocess communications. This CSP model let us check if the overall system was deadlock-free and if there was no concurrent processing of security-critical functions.

These checks were carried out automatically, using Formal Systems Europe's failures-divergence refinement tool. This helped us find significant flaws in our first design and gave us much greater confidence in the final design.

Furthermore, implementing the design using Ada95 tasks, rendezvous, and protected objects was straightforward. We devised rules for systematically translating CSP into code. This successfully yielded code that worked the first time, a rare experience among concurrent programmers.

Programming languages and static analysis

Given the formality of the specification and design, we hoped to carry this through into the implementation. ITSEC and Common Criteria require the use of programming languages with "unambiguous meaning," yet the use of formal implementation languages remains rare. Experience suggests that, despite sound protocols and cryptography, sloppy implementation remains a common source of failure in supposedly secure systems—the ubiquitous "buffer overflow" attack, for instance.

A cure for sloppy implementation is a formal implementation language for which we can carry out static analysis—that is, analyzing program properties such as information flow without actually running the program.

To enable static analysis to produce useful results, the language must be as precise as possible. In the presence of ambiguity, static analysis must make assumptions (for example, "The compiler evaluates expressions left-to-right") that can render the results dubious. Alternatively, it might attempt to cover all possible outcomes of any ambiguity—this leads to an explosion in analysis time that makes the tool unusable.

To enable static analysis to produce useful results, the language must be as precise as possible.

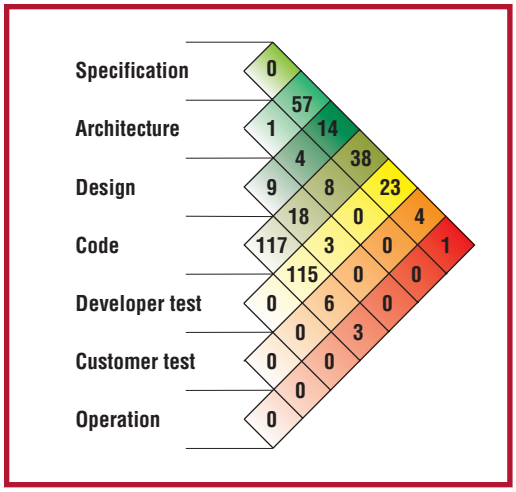


Figure 2. Life-cycle phases where defects were introduced and where they were detected and removed.

Table 1

Distribution of effort.

Activity	Effort (%)
Requirements	2
Specification and architecture	25
Code	14
Test	34
Fault fixing	6
Project management	10
Training	3
Design authority	3
Development- and target-environment	3

Unfortunately, few languages meet this need. Trends in programming-language design have favored dynamic behavior (for example, late binding of function calls in object-oriented languages) and performance (for example, unchecked arithmetic and array access in C) over safety. These features are dramatically at odds with the needs for static analysis, and so they are inappropriate for constructing high-integrity systems. The safety-critical community, where the use of high-integrity subset languages such as Spark is the norm, has broadly accepted this.

Borrowing from Ross Anderson's well-known analogy,[11] clearly, if you're programming Satan's computer, you should not use Satan's programming language!

The use of Spark in the CA

In the CA, we used an information flow–centered software architecture. This maximizes cohesion and minimizes coupling between units. We carefully chose between Spark and Ada95 for each compilation unit, on the basis of the required separation between security-related functions in the system. Even though an Ada compiler processes both these languages, it was worth regarding them as separate languages for the purposes of the design.

All Spark code had to pass through the Spark Examiner with no unjustified warnings or errors before any other review or inspection activity. This let reviewers focus on important topics (such as "Does this code implement the FTLS?") rather than worrying about more trivial matters such as dataflow errors or adherence with coding standards.

We used only the most basic form of annotation and analysis the Examiner offered. We did not perform proof of partial correctness of the code. We did carry out some proofs of exception freedom.

Informally, the defect rate in the Spark code was extremely low. Spark programs have an uncanny habit of simply running the first time. The most significant difficulties arose in the system's more novel parts, especially in areas that we had not formally designed or specified, such as the manipulation of Win32 named pipes, the database interface, and the handling of machine failures.

Results

Overall, the development has been successful. The number of system faults is low compared with systems developed using less formal approaches.[12] The delivered system satisfies its users, performs well, and is highly reliable. In the year since acceptance, during which the system was in productive use, we found four faults. Of course, we corrected them as part of our warranty. This rate, 0.04 defects per KLOC, is far better than the industry average for new developments.[12]

Figure 2 shows the life-cycle phases where defects were introduced and where they were detected and removed. For example, 23 errors were introduced at the specification phase and removed during developer test. A good development method aims to find errors as soon as possible after they are introduced, so the numbers on the right of Figure 2 should be as small as possible.

The delivered system contained about 100,000 lines of code. Overall productivity on the development—taking into account all project activities, including requirements, testing, and management—was 28 lines of code per day. The distribution of effort shows clearly that fault fixing constituted a

relatively small part of the effort (see Table 1); this contrasts with many critical projects where fixing of late-discovered faults takes a large proportion of project resources.

Three significant conclusions that we can draw from this work concern the use of COTS for secure systems, the practicality of formal methods, and the choice of programming language.

You can build a secure system using insecure components, including COTS. The system's overall architecture must guarantee that insecure components cannot compromise the security requirements. This resembles the approach taken to safety-critical systems.

Using formal methods, as required by the higher levels of ITSEC and the Common Criteria, is practical. Our experience in this and other projects show that well-considered use of formal methods is beneficial. Of course, neither formal methods nor any other known method can completely eliminate defects. For example, we didn't discover a few specification errors until user test. We can attribute these to incorrect formalization of the detailed requirements. Nevertheless, formal methods do reduce the number of late-discovered errors and, thus, the overall system cost.

Similarly, Spark is certainly not a magic bullet, but it has a significant track record of success in the implementation of high-integrity systems. Spark, we believe, is unique in actually meeting the implementation requirements of the ITSEC and CC schemes. Spark's support for strong static analysis and proof of program properties (for example, partial correctness or exception freedom) means that you can meet the CC requirements for formal development processes. The language subset's simplicity and the data- and information-flow analysis offered by the Examiner make a large class of common errors simply impossible to express in Spark.

Acknowledgments

We thank John Beric of Mondex International for his comments on an early draft of this article. The SPARK programming language is not sponsored by or affiliated with SPARC International and is not based on the SPARC architecture.

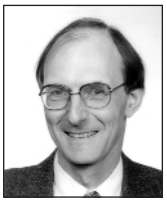
References

1. M. Hendry, *Smartcard Security and Applications*, 2nd ed., Artech House, Norwood, Mass., 2001.
2. *Provisional Harmonised Criteria*, version 1.2, Information Technology Security Evaluation Criteria (ITSEC), Cheltenham, UK, June 1991.
3. *ISO/IEC 15408:1999, Common Criteria for Information Technology Security Evaluation*, version 2.1, Int'l Organization for Standardization, Geneva, 1999; www.commoncriteria.org (current Nov. 2001).
4. J. Hammond, R. Rawlings, and A. Hall, "Will It Work?" *Proc. Fifth Int'l Symp. Requirements Eng.* (RE 01), IEEE CS Press, Los Alamitos, Calif., 2001, pp. 102–109.
5. S. Robertson and J. Robertson, *Mastering the Requirements Process*, Addison-Wesley, Reading, Mass., 1999.
6. D. Coleman et al., *Object-Oriented Development: The Fusion Method*, Prentice-Hall, Upper Saddle River, N.J., 1994.
7. J.M. Spivey, *The Z Notation: A Reference Manual*, 2nd ed., Prentice-Hall, Upper Saddle River, N.J., 1992.
8. J. Barnes, *High Integrity Ada: The SPARK Approach*, Addison-Wesley, Reading, Mass., 1997.
9. S. King et al., "Is Proof More Cost-Effective Than Testing?" *IEEE Trans. Software Eng.*, vol. 26, no. 8, Aug. 2000, pp. 675–686.
10. *CESG Computer Security Manual "F": A Formal Development Method for High Assurance Systems*, Communications Electronics Security Group, Cheltenham, UK, 1995.
11. R.J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, John Wiley & Sons, New York, 2001.
12. S.L. Pfleeger and L. Hatton, "Investigating the Influence of Formal Methods," *Computer*, vol. 30, no. 2, Feb. 1997, pp. 33–43.

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/publications/dlib.

Spark is certainly not a magic bullet, but it has a significant track record of success in the implementation of high-integrity systems.

About the Authors



Anthony Hall is a principal consultant with Praxis Critical Systems. He is a specialist in requirements and specification methods and the development of software-intensive systems. He has been a keynote speaker at the International Conference on Software Engineering, the IEEE Conference on Requirements Engineering, and other conferences. He has an MA and a DPhil from Oxford University. He is a fellow of the British Computer Society and a Chartered Engineer. Contact him at Praxis Critical Systems Ltd., 20 Manvers St., Bath BA1 1PX, UK; anthony.hall@praxis-cs.co.uk.

Roderick Chapman is a software engineer with Praxis Critical Systems, specializing in the design and implementation of high-integrity real-time and embedded systems. He has also been involved with the development of the Spark language and its associated static-analysis tools. He received an MEng in computer systems and software engineering and a DPhil in computer science from the University of York. He is a member of the British Computer Society and is a Chartered Engineer. Contact him at Praxis Critical Systems Ltd., 20 Manvers St., Bath BA1 1PX, UK; rod.chapman@praxis-cs.co.uk.