Chapter 17
The Tokeneer Experiments

Jim Woodcock, Emine Gökçe Aydal, and Rod Chapman

For Tony Hoare, to celebrate his 75th birthday.

Abstract We describe an experiment conducted as part of a pilot project in the Verified Software Initiative (VSI). We begin by recounting the background to the VSI and its six initial pilot projects, and give an update on the current progress of each project. We describe one of these, the Tokeneer ID Station in greater detail. Tokeneer was developed by Praxis High Integrity Systems and SPRE for the US National Security Agency, and it has been acclaimed by the US National Academies as representing best practice in software development. To date, only five errors have been found in Tokeneer, and the entire project archive has been released for experimentation within the VSI. We describe the first experiment using the Tokeneer archive. Our objective is to investigate the dependability claims for Tokeneer as a security-critical system. Our experiment uses a model-based testing technique that exploits formal methods and tools to discover nine anomalous scenarios. We discuss four of these in detail.

17.1 Introduction (by JW)

In 2002, I attended a colloquium in Lisbon to celebrate the UN Software Technology Institute's 10th Anniversary [1]. Tony Hoare gave a talk on the use of assertions in current Microsoft practice, where they instrument programs as software testing

J. Woodcock (✉)
Department of Computer Science, University of York, Heslington,
York YO10 5DD, Great Britain
e-mail: jim@cs.york.ac.uk

E.G. Aydal
Department of Computer Science, University of York
e-mail: aydal@ieee.org

R. Chapman
Altran Praxis Limited, 20 Manvers Street, Bath BA1 1PX, Great Britain
e-mail: rod.chapman@altran-praxis.com

C.B. Jones et al. (eds.), *Reflections on the Work of C.A.R. Hoare*,
DOI 10.1007/978-1-84882-912-1_17,

probes [40]. He went on to describe the greater benefits that would follow from using a tool to check a program's adherence to its assertions: *the Verifying Compiler*. He concluded by saying that building a verifying compiler would be a "splendid opportunity for academic research" and "one of the major challenges of Computing Science in the twenty-first century," likening it to the Human Genome project.

Tony had invited me to lecture at that year's Marktoberdorf Summer School. During one of the excursions, he repeated to me his idea for the verifying compiler as a way of galvanising the computer science community into a productive long-term research programme: a Grand Challenge. This, Tony said, was the dream of Jim King's doctoral thesis [59], a dream abandoned in the 1970s as being well beyond current theorem-proving capabilities. But much progress had been made in the following 30 years, both in hardware capacity and in the software technologies for verification. Tony suggested that the renewed challenge of an automatic verifying compiler could provide a focus for interaction, cross-fertilisation, advancement, and experimental evaluation of all relevant verification technologies. Perhaps by a concerted international effort, we might be able to meet this challenge within 15–25 years. I was now recruited to the cause.

In November that year, there was a UK Grand Challenge Workshop in Edinburgh, where more than 100 proposals were submitted [43]. These proposals were distilled into just seven grand challenges, one of which included the verifying compiler: *GC6—Dependable Systems Evolution*. As its name suggests, this challenge was based on a very broad understanding of software correctness, and tried to include as wide a community of researchers as possible, spanning the range of interests from full functional correctness through to issues of dependability, where formalisation is difficult, if at all possible. Three threads of activity were launched to progress GC6: software verification, dependability, and evolution. Tony Hoare, Cliff Jones, and Brian Randell tried to maintain the breadth of the grand challenge by emphasising the importance of the work on dependability and evolution [44], but proposals like this are shaped by the availability and enthusiasm of the individuals involved, and the only thread that has so far really taken off was the one inspired by Tony's original idea of the verifying compiler. The main activity within this thread has been experimental work on pilot verification projects, as reported in this paper.

The term "verifying compiler" is often misunderstood by researchers, who sometimes hear "verified compiler." It is also often thought of as just a single tool for a single programming language, probably an idealised academic one at that. But this was never Tony's intention. The verifying compiler was a cipher for an integrated set of tools checking correctness, in a very broad sense, of a wide range of programming artefacts. In promoting the grand challenge, Tony talked about things that might have surprised his colleagues only a few years before. He talked not just about full functional correctness, but about checking isolated properties and about the subtler notions of robustness and dependability. He talked about tools that were neither sound nor complete, about inter-operability of tools, and about the practical programming languages used in industry. He even talked about testing.

GC6 has now transformed into an international grand challenge: the Verified Software Initiative, led by Tony Hoare and Jay Misra, and its manifesto [42]

represents a consensus position that has emerged from a series of national and international meetings, workshops, and conferences. Overviews of the background and objectives may be found in [52, 77, 78]. Surveys of the state of the art are available [41], covering practice and experience in formal methods [8, 80], automated deduction for verification [68], and software model checking [51].

Interest in the VSI's research agenda has grown from just a few dedicated individuals in 2002 to a distinct community today. There are 55 different international research groups working in the experimental strand alone. Many members of this community are young researchers, making important contributions at early stages of their career. They have their own conference series, Verified Software: Theories, Tools, and Experiments (Zurich 2005 [64], Toronto 2008 [69], and Edinburgh 2010). They have published a series of special journal issues (some are still in press): ACM Computing Surveys, Formal Aspects of Computing [18, 54], Science of Computer Programming [28], Journal of Object Technology, Journal of Universal Computer Science [4], and Software Tools for Technology Transfer. They organise working meetings at leading and specialist conferences: FM Symposium, FLoC, SBMF, ICTAC, ICFEM, ICECCS, and SEFM. They represent six continents: North and South America, Europe, Asia, Australia and Africa.

17.2 The Verified Software Repository

The main focus of the UK's contribution to the VSI is on building a Verified Software Repository [9, 75], which will eventually contain hundreds of programs and components, amounting to several million lines of code. This will be accompanied by full or partial specifications, designs, test cases, assertions, evolution histories, and other formal and informal documentation. Each program will be mechanically checked by at least one tool, although most will be analysed by a series of tools in a comparative study. The Repository's programs are selected by the research community as realistically representing the wide diversity of computer applications, including smartcards, embedded software, device drivers, a standard class library, an embedded operating system, a compiler for a useful language (possibly Java Card), and parts of the verifier itself, a program generator, a communications protocol (possibly TCP/IP), a desk-top application, parts of a web service (perhaps Apache). The main purpose of the Repository is to advance science, but reusable verified components may well be taken up in real-life application domains.

Verification of repository components already includes the wide spectrum of program properties, from avoidance of specific exceptions like buffer overflow, general structural integrity (crash-proofing), continuity of service, security against intrusion, safety, partial functional correctness, and (at the highest level) total functional correctness [45]. The techniques used are similarly wide ranging: from unit testing to partial verification, through bounded model checking to fully formal proof. To understand exactly what has been achieved, each claim for a specific level of correctness is accompanied by a clear informal statement of the assumptions and

limitations of the proof, and the contribution that it makes to system dependability. The progress of the project can be measured by the automation involved in reaching each level of verification for each module in the Repository. Since the ultimate goal of the project is scientific, the ultimate aim is for complete automation of every property, higher than the expectations of a normal engineer or customer.

In the remainder of this introductory section, we describe the status of some early pilot projects that are being used to populate the Repository. Mondex is a smartcard for electronic finance. The Verified Filestore is inspired by a real space-flight application. FreeRTOS is a real-time scheduler that is very widely used in embedded systems. The Cardiac Pacemaker is a real system, and is representative of an important class of medical devices. Microsoft's Hypervisor is based on one of their future products. Finally, Tokeneer is a security application involving biometrics. These six pilot projects encompass a wide variety of application areas and each poses some important challenges for verification.

17.2.1 Mondex

The following description is based on [81]. In the early 1990s, the National Westminster Bank and Platform Seven (a UK software house) developed a smartcard-based electronic cash system, Mondex, suitable for low-value cash-like transactions, with no third-party involvement, and no cost per transaction. A discussion of the security requirements can be found in [73,81]; a description of some wider requirements can be found in [2]. It was crucial that the card was secure, otherwise money could be electronically counterfeited, so Platform Seven decided to certify Mondex to one of the very highest standards available at the time: ITSEC Level E6 [46], which approximates to Common Criteria Level EAL7 [14] (see the discussion in Section 17.3). This mandates stringent requirements on software design, development, testing, and documentation procedures. It also mandates the use of formal methods to specify the high-level abstract security policy model and the lower-level concrete architectural design. It requires a formal proof of correspondence between the two, in order to show that the concrete design obeys the abstract security properties. The evaluation was carried out by the Logica Commercial Licensed Evaluation Facility, with key parts subcontracted to the University of York to ensure independence.

The target platform smartcard had an 8-bit microprocessor, a low clock speed, limited memory (256 bytes of dynamic RAM, and a few kilobytes of slower EEPROM), and no built-in operating system support for tasks such as memory management. Power could be withdrawn at any point during the processing of a transaction. Logica was contracted to deliver the specification and proof using Z [71, 79]. They had little difficulty in formalising the concrete architectural design from the existing semi-formal design documents, but the task of producing an abstract security policy model that both captured the desired security properties (in particular, that "no value is created" and that "all value is accounted for") and

provably corresponded to the lower-level specification, was much harder. A very small change in the design would have made the abstraction much easier, but was thought to be too expensive to implement, as the parallel implementation work was already well beyond that point. The 200-page proof was carried out by hand, and revealed a small flaw in one of the minor protocols; this was presented to Platform Seven in the form of a security-compromising scenario. Since this constituted a real security problem, the design was changed to rectify it. The extensive proofs carried out were done manually using some novel techniques [72]. The decision not to use mechanical theorem proving was intended to keep costs under control. Recent work (reported below) has shown that this was overly cautious, and that Moore's Law has swung the balance further in favour of cost-effective mechanical verification.

In 1999, Mondex achieved its ITSEC Level E6 certificate: the very first product ever to do so. As a part of the ITSEC E6 process, the entire Mondex development was additionally subjected to rigorous testing, which was itself evaluated. No errors were found in any part of the system subjected to the use of formal methods.

Mondex was revived in 2006 as a pilot project for the Grand Challenge in Verified Software. The main objective was to test how the state of the art in mechanical verification had moved on in 10 years. Eight groups took up the challenge using the following formal methods (with references to a full discussion of the kinds of analysis that were performed in each case): Alloy [67], ASM [39], Event-B [10], OCL [60], PerfectDeveloper,[1] π-calculus [53], Raise [36], and Z [32]. The cost of mechanising the Z proofs of the original project was 10% of the original development cost, and so did not dominate costs as initially believed. Interestingly, almost all techniques used in the Mondex pilot achieved the same level of automation, producing similar numbers of verification conditions and requiring similar effort (see [54] for a discussion of these similarities).

17.2.2 Verified Filestore

At an early workshop on the Verifying Compiler, Amir Pnueli suggested that we should choose the verification of the Linux kernel as a pilot project. It would be a significant challenge, and would have a lasting impact. Joshi and Holzmann suggested a more modest aim: the verification of the implementation of a subset of the POSIX filestore interface suitable for flash-memory hardware with strict fault-tolerance requirements to be used by forthcoming NASA missions [56]. They required the system would prevent corruption in the presence of unexpected power-loss, and that it would be able to recover from faults specific to flash hardware (e.g., bad blocks, read errors, bit corruption, wear-levelling, etc.) [35] . The POSIX file-system interface [55] was chosen for four reasons: (i) it is a clean, well-defined,

[1] No paper is available on the PerfectDeveloper treatment of Mondex, but see [24] for a general discussion of the PerfectDeveloper tool itself.

and standard interface that has been stable for many years; (ii) the data structures and algorithms required are well understood; (iii) although a small part of an operating system, it is complex enough in terms of reliability guarantees, such as unexpected power-loss, concurrent access, or data corruption; and (iv) modern information technology is massively dependent on reliable and secure information availability. An initial subset of the POSIX standard has been chosen for the pilot project. There is no support for: (i) file permissions; (ii) hard or symbolic-links; or (iii) entities other than files and directories (e.g., pipes and sockets). Adding support for (i) is not difficult and may be done later, whereas support for (ii) and (iii) is more difficult and might be beyond the scope of the challenge. Existing flash-memory file-systems, such as YAFFS2 [50], do not support these features, since they are not usually needed for the functionality of an embedded system.

Freitas and Woodcock have mechanically verified existing Z models of the POSIX API [33] and a higher-level transaction processing API [31, 34]. Freitas has shown how to verify datatypes for the design of operating system kernels [30]. Butterfield, Freitas, and Woodcock have modelled the behaviour of flash memory devices [11–13]. Butler has specified a tree-structured file system in Event-B [26], and has specified some of the details of the flash file system itself [25]. Mühlberg and Lüttgen have used model checking to verify compiled file-system code [65], and Jackson has used Alloy to produce a relational model of aspects of a flash file system [57]. Ferreira and Oliveira have integrated Alloy, VDM++, and HOL into a tool chain to verify parts of the Intel flash file system [29]. Finally, Kim has explored the flash multi-sector read operation using concolic testing [58].

17.2.3 FreeRTOS

Richard Barry (Wittenstein High Integrity Systems) has proposed the correctness of their open-source real-time mini-kernel as a pilot project. It runs on a wide range of different architectures and is used in many commercial embedded systems. There are over 5,000 downloads per month from SourceForge, putting it in the top 250 of SourceForge's 170,000 codes. It is less than 2,500 lines of pointer-rich code, which makes it small, but very interesting. The first challenge is to analyse the program for structural integrity properties, for example, to prove that its elaborate use of pointers is safe. The second challenge is to make a rational reconstruction of the development of the program, starting from an abstract specification, and refining down to working code, with all verification conditions discharged with a high level of automation. These challenges push the current state of the art in both program analysis and refinement of pointer programs.

Déharbe has produced an abstract specification of FreeRTOS [27] and Machado has shown how to generate tests automatically from the code [63]. Craig is working on the formal specification and refinement in Z of more general operating system kernels [22, 23]. Some of his models have been verified by Freitas and Woodcock.

17.2.4 Cardiac Pacemaker

Boston Scientific has released into the public domain the system specification for a previous generation pacemaker, and is offering it as a challenge problem. They have released a specification that defines functions and operating characteristics, identifies system environmental performance parameters, and characterises anticipated uses. This challenge has multiple dimensions and levels. Participants may choose to submit a complete version of the pacemaker software, designed to run on specified hardware, they may choose to submit just a formal requirements documents, or anything in between. McMaster University's Software Quality Research Laboratory is putting in place a certification framework to simulate the concept of licensing. This will enable the Challenge community to explore the concept of licensing evidence and the role of standards in the production of such software. Furthermore, it will provide a more objective basis for comparison between putative solutions to the Challenge.

Lawson and his colleagues at McMaster University maintain a web page describing the state of the Pacemaker pilot project [61]. It gives details of the pacemaker hardware reference platform, developed by students at the University of Minnesota, based on an 8-bit PIC18F4520 microcontroller. Macedo, Fitzgerald and Larsen have an incremental development of a distributed real-time model of a cardiac pacing system using VDM [62]. Gomes and Oliveira have specified the Pacemaker in Z, and carried out proofs of consistency of their specification using ProofPowerZ [37].

17.2.5 Microsoft Hypervisor

Schulte and Paul initiated work within Microsoft on a hypervisor (a kind of separation kernel), and it has been proposed by Thomas Santen as a challenge project. The European Microsoft Innovation Center is collaborating with German academic partners and the Microsoft Research group for Programming Languages and Methods on the formal verification of the new Microsoft Hypervisor, to be released as part of as new Windows Server. The Hypervisor will allow multiple guest operating systems to run concurrently on a single hardware platform. By proving the mathematical correctness of the Hypervisor, they will control the risks of malicious attack. Cohen has briefly described the Microsoft Hypervisor project [17].

17.3 Pilot project: Tokeneer ID Station

In this section, we describe one of the pilot projects in a lot more detail: the Tokeneer ID Station (TIS), a project conducted by Praxis High Integrity Systems and SPRE for the US National Security Agency. See [15] for an overview of the system, and [6] for an account of how it was engineered. Tony Hoare has already recorded his opinion of the work carried out: "The Tokeneer project is a milestone in the transfer of

program verification technology into industrial application" [74]. A report from the US National Academies [48] refers to several Praxis projects as examples of best practice in software engineering, particularly in the areas of formal methods and programming language design and verification.

The Tokeneer project was originally conceived to supply evidence about whether it is economically feasible to develop systems that can be assured to the higher levels of the Common Criteria Security Evaluation, the ISO/IEC 15408 standard for computer security certification [14]. The standard defines seven levels for evaluating information technology security:

- EAL7: formally verified design and tested
- EAL6: semi-formally verified design and tested
- EAL5: semi-formally designed and tested
- EAL4: methodically designed, tested, and reviewed
- EAL3: methodically tested and checked
- EAL2: structurally tested
- EAL1: functionally tested

Barnes et al. report that an evaluation in 1998 to what is now understood as EAL4 cost about US$2.5 million [6].

Numerous smartcard devices have been evaluated at EAL5, as have multilevel secure devices such as the Tenix Interactive Link. XTS-400 (STOP 6) is a general-purpose operating system, which has been evaluated at an augmented EAL5 level. An example of an EAL6 certified system is the Green Hills Software INTEGRITY-178B operating system, the only operating system to achieve EAL6 so far. The Tenix Interactive Link Data Diode Device has been evaluated at EAL7 augmented, the only product to achieve this.

The problem with these higher levels of the Common Criteria is that industry believes that it is simply too expensive to develop systems to this standard. The argument is a familiar one. In 1997, the UK Government Communications Headquarters held a workshop to discuss the view held in industry that it was too expensive to use formal methods to achieve ITSEC Level E6 [76], approximately EAL7. The Mondex project (see Section 17.2) provided evidence to the contrary.

So the objective of the Tokeneer project was to explore the feasibility of developing cost-effective, high-quality, low-defect EAL5 systems, and to provide evidence for both EAL6 and EAL7. It was a rare and valuable opportunity to undertake the controlled measurement of productivity and defect rates. Remarkably, the entire project archive is openly available and may be downloaded from [74].

Praxis have a well-developed software engineering method that addresses not only assurance, but also cost requirements. Their method starts from requirements analysis using their REVEAL technique, continues with specification and development, using formal methods where appropriate, until an implementation is reached in SPARK, a high-level programming language and toolset designed for writing software for high-integrity applications [7]. They have a successful record of using their method to develop commercial applications of formal methods, with costs reportedly lower than traditional manual object-oriented methods.

Tokeneer was the subject of an earlier NSA research project investigating the use of biometrics for physical access control to a secure room containing user workstations (the *enclave*). The Tokeneer ID Station contributed to a further development of the original system. The key idea is that users have smartcard security tokens that must be used both to gain access to the enclave and to use the workstations once the user is inside. There are smartcard and biometric readers outside the enclave; if a user passes their identity tests, then the door opens for entry. Authorisation information is written onto the card for subsequent workstation access. This information describes privileges the user can enjoy for this visit, including times of working, security clearance and user roles.

In what follows, it is important to understand the Tokeneer ID Station security target, in order to answer the question, "Is Tokeneer *really* secure?" The requirements assume that the enclave is situated in a high-security area, and so all the users will have passed a stringent security clearance procedure, either as NSA employees or as accredited visitors. As a consequence, it may be safely assumed that no user will ever attempt a malicious attack on the enclave. Instead, the security measures are intended to prevent accidents: *unintentional, unauthorised access* to the enclave and the data provided by its workstations.

The overall functionality of the Tokeneer ID Station was formalised in a 100-page Z specification. The code was developed in two parts. The core security-related functionality was implemented in SPARK, and amounts to 9,939 lines of code, with 6,036 lines of flow annotations, 1,999 lines of proof annotations, and 8,529 lines of comments. The remainder of the system was not security critical, and so was developed using Ada95, comprising 3,697 lines of code, no flow or poof annotations, and 2,240 lines of comments. The entire development required 260 man-days, provided by three people working part-time over 9 months.

The task set by NSA was to conform to EAL5. The development actually exceeded EAL5 requirements in several areas, including configuration control, fault management, and testing. The main body of the core development work was carried out to EAL5. But the specification, design, implementation and proof of correspondence were conducted to EAL6 and 7. So why would Praxis do more than they were asked to do? Because they were told that, if they could produce evidence at these higher levels within budget, then they should.

The Tokeneer project archive has been downloaded many hundreds of times. Knight [38] has verified Tokeneer properties using the PVS theorem prover [66]. Jackson is working to broaden the range of properties of SPARK programs that are automatically verifiable, thus speeding up verification and supporting use of richer assertions [49]. Work is underway to re-implement Tokeneer using the PerfectDeveloper system (see [24] for details of PerfectDeveloper). Aydal and Woodcock have been analysing the system to search for attacks [3], work reported below.

But how good was the original development of Tokeneer? In fact, only five defects have been found in Tokeneer since it was deployed within NSA in 2004. We describe each of the defects, reflecting on their causes and significance.

17.3.1 Defect 1

This account is based on that in [15, Section 17.3]. When the Tokeneer code was re-analysed in August 2008, in preparation for the public release of the entire archive, the tool that summarised the proof obligations (the POGS tool) revealed a single undischarged verification condition. Further investigation showed this to be in the subprogram `ConfigData.ValidateFile.ReadDuration`. The code in question concerns validation of an integer value that is read from a file, but is expected to be in the range 0–200 s before it is converted into a number of tenths of seconds in the range 0–2000. The offending undischarged VC is essentially:

```
H1:  rawduration__1 >= - 2147483648 .
H2:  rawduration__1 <= 2147483647 .
  ->
C1:  success__1 ->
       rawduration__1 * 10 >= - 2147483648 and
       rawduration__1 * 10 <= 2147483647 .
```

The code is from line 222 of `configdata.adb`:

```
if Success and then
  (RawDuration * 10 <= Integer(DurationT'Last) and
   RawDuration * 10 >= Integer(DurationT'First)) then
```

This VC clearly has a counterexample. For instance, when `RawDuration` $= 10^9$, `H1` and `H2` are true, but `C1` is false. This reflects the possibility of an integer overflow when multiplying by 10 before the range of `RawDuration` is checked. The correction to the code is trivial. If replaced by:

```
if Success and then
  (RawDuration <= Integer(DurationT'Last) / 10 and
   RawDuration >= Integer(DurationT'First) / 10) then
```

then all VCs discharge successfully.

Why was this defect not discovered and reported during the original development? The original project used the SPARK Examiners "`rtc`" switch to generate VCs, which it does for partial correctness and run-time errors, but *omits* those side-conditions relating to Ada's `Overflow_Check`. Previously, the SPARK toolset was limited in its capability to discharge these VCs, so these were omitted from the original project. Subsequently, the SPARK toolset has become far more capable with regard to overflow conditions, through the use of the compiler-dependent configuration file, and the base-type assertion for integer types. Users can now generate VCs using the "`vcg`" switch, which does include VCs for overflow checks. It is interesting to note that this defect was not discovered by any testing during the original project, or any use or attempt to analyse the system since the initial delivery.

What about the security impact? First, there is a potential denial-of-service attack resulting from this defect: a malicious user holding the "security officer" role can deliberately terminate the TIS core software by supplying a malformed configuration data file, rendering the system unusable. More seriously, the software can be terminated in this fashion with the enclave door open.

17.3.2 Defect 2

The first defect was discovered by Spinellis in October 2008; see his blog [70], where he reports the following. The Tokeneer function `SystemFaultOccurred` is required to return true exactly when a critical system fault has occurred while attempting to maintain the audit log. The code that implements this uses a global variable, `AuditSystemFault`, which is set to true whenever a fault is detected. Spinellis lists all the assignments to `AuditSystemFault` that he found in the code (`OK` is a variable set by various system functions).

```
AuditSystemFault := AuditSystemFault or not OK;
AuditSystemFault := AuditSystemFault or not OK;
AuditSystemFault := AuditSystemFault or not OK;
AuditSystemFault := AuditSystemFault or not OK;
AuditSystemFault := True;
AuditSystemFault := True;
AuditSystemFault := True;
AuditSystemFault := True;
AuditSystemFault := not OK;
AuditSystemFault := AuditSystemFault and not OK;
```

But there is an anomaly in the last assignment, which is used when a log file is deleted. The conjunction used instead of a disjunction has the effect of clearing the `AuditSystemFault` flag if the deletion is successful, and failing to set it if the deletion fails, but no fault was detected before. Spinellis found this bug by inspection in less than an hour of browsing. In fact, it was in the second file he looked at, the first being very short.

17.3.3 Defect 3

The second defect was found by the CodePeer tool on or about 24 August 2009. CodePeer (developed jointly by AdaCore and SofCheck) statically analyses Ada programs for a wide range of flaws, including: pointer misuse, buffer overflows, numeric overflow or wraparound, division by zero, dead code, unused variables, and race conditions. The tool detected the following error.

The procedure `KeyStore.DoFind` contains the following code sequence:

```
Interface.FindObjectsInit(Template    => Template,
                          ReturnValue => RetValIni);
if RetValIni = Interface.Ok then
   Interface.FindObjects(HandleCount   => HandleCount,
                         ObjectHandles => Handles,
                         ReturnValue   => RetValDo);
  if RetValIni = Interface.Ok then
     Interface.FindObjectsFinal(ReturnValue =>
                                RetValFin);
  end if;
end if;
```

The test in the second conditional statement is wrong: it should be

```
if RetValDo = Interface.Ok then
```

of course. This is almost certainly a cut-and-paste error from the enclosing conditional statements, but why was it not detected during the original verification process? The procedure call above the offending test assigns to `RetValDo`, but unfortunately, there is a meaningful reference to `RetValDo` later in the subprogram, so the SPARK flow-analyser fails to spot this as an ineffective assignment, as might have been expected. The offending code also gives rise to several dead paths through the code, since there are paths with traversal condition

```
(RetValIni = Interface.Ok) and
-- then branch of outer if statement
(RetValIni /= Interface.Ok)
-- erroneous else branch of inner if statement
```

But the verification conditions arising from this code are all trivially true, since this procedure has an implicitly true postcondition. Therefore, the SPARK Simplifier does not bother with proof-by-contradiction and so fails to spot the dead paths.

The discovery of this error has led to the development of a new tool, *ZombieScope*, to detect dead code. It is similar to the SPARK Simplifier, except that but it looks only for contradictory hypotheses, ignoring the conclusions of all VCs. Any VCs that are found to have contradictory hypotheses are flagged up as indicating dead paths. The security impact of this bug is not known, as it has yet to be analysed closely enough.

17.3.4 Defect 4

The third defect was found using the most recent GNAT compiler, a free, high-quality, complete compiler for Ada95, integrated into the GCC compiler system.

The defect was found in `AuditLog.AddElementToFile`. `NameOfType`, using the `-gnatwa` flag (all warnings mode). The offending code is:

```
function NameOfType (E : AuditTypes.ElementT )
return ElementTextT
is
  --# hide NameOfType;
  ElementText : ElementTextT := NoElement;
begin
  ElementText(1..
          AuditTypes.ElementT'Image(ElementID)'Last)
    := AuditTypes.ElementT'Image(ElementID);
  return ElementText;
end NameOfType;
```

GNAT reports that the formal parameter `E` is not referenced, which is of course quite right: there is no reference to `E` in the body of that function. The reference to `ElementID`, which is a global variable that is visible from that scope, should be `E` here, so it should read:

```
ElementText(1..
  AuditTypes.ElementT'Image(E)'Last)
    := AuditTypes.ElementT'Image(E);
```

Why was this defect not found before? It would have been difficult to detect the error during development using the tools available at the time. When the code was first written, the much earlier version of GNAT used did not implement this warning. The code in question is not even SPARK, but actually Ada, since it uses a feature that is not part of SPARK (an array-slice in the assignment), and the code has to be hidden from the analyser, which would otherwise reject it. Consequently no flow analysis was conducted at all. The original decision to hide this code was almost certainly a mistake. Bugs like this can creep in without the rigour of the SPARK tools. This bug was also missed in code-review, suggesting that the reviewing of hidden units should have been given more attention. If the code were SPARK, the tools would certainly have spotted it: an unreferenced formal parameter is always reported by the SPARK Examiner.

What is the impact on correctness and security? Curiously, none. There is exactly *one* call to this function, which reads:

```
File.PutString( TheFile => TheFile,
                Text    => NameOfType (ElementID),
                Stop    => 0 );
```

So, in this single call `E` (the formal parameter) is synonymous with `ElementID` (the global variable), so there is no foul. But it is a bug waiting to happen if this code were ever called again with a different parameter, and the developers would fix it given the chance, so it still qualifies as a defect [16].

17.3.5 Defect 5

The fourth defect was also found by SofCheck's CodePeer tool, in September 2009, in `TokenReader.Poll.CheckCardState`. CodePeer reports that the final branch of the case statement

```
when Interface.InvalidCardState =>
  MarkTokenBad;
```

is dead—telling us that `CardState` can never have the value

```
InvalidCardState
```

CodePeer (through some surprisingly clever inter-procedural value propagation) is able to determine that the `RawCardState` value returned from

```
TokenReader.Interface.Status
```

is always in the range 1–6, not 0–6. This can be verified by inspection by following the sequence of calls down the call tree into the support software (in `support/tokenapi.adb`). The SPARK tools did not detect this, since the dead-path analysis is *intra*-procedural and based on the contracts of the called units alone.

Is this a bug? Well, not really, but it is an interesting observation that the analysis of the use of subtypes in the code could be improved.

17.4 A Token Experiment

In this section, we report on an experiment that we performed on Tokeneer. Our motivation was to take a system that has been developed using best practice and to see if there is anything more that we can say about it. In particular, since it is a security-critical system, can we break it? One way of proceeding in our experiment would have been to search for undischarged verification conditions and proof obligations, hoping to find that at least one of them would turn out not to be true.

But a second motivation for the experiment was to revisit the original goals of GC6, and to make a small contribution towards understanding not just the functional correctness of Tokeneer *(Does it correctly implement its specification?)*, but to say something about its dependability *(Is it really secure?)*. For this reason, we decided to try to validate the system against its requirements, and in particular, the security requirement of no accidental access to the enclave or to its workstations.

To do this, we used a novel model-based testing technique that exploits formal methods and tools: assertion-guided model-based robustness testing. The main hypothesis of this study was:

> Applying robustness testing in a model-based manner with the use of a separate test model may reveal requirements-related faults that may not necessarily be detected by formal verification techniques.

In addition to this hypothesis, we assumed two test hypotheses that helped us define the test oracle and the test selection algorithm. These hypotheses are valid for very specific kinds of system, and their usefulness must be judged by the quality of the results: have they uncovered any genuine failures? We revisit this point at the end of the paper. We consider first the *Redundant Models Hypothesis*:

> If there are two different specifications of the same System under Test (SuT) that conform to the same set of requirements, then their fault domains must match with respect to some test set T.

By using two models of the same system that are independently produced from the requirements of the SuT, and a test suite generated by using one of these models, we tried to reveal faults within these models by finding inconsistent behaviours in their fault domains. Having one model for code generation and a separate model for test case generation not only introduces the redundancy required for the testing process, but also separates concerns whilst producing these models. Additionally, there may be situations where the design model of the system may not be available due to confidentiality reasons (e.g., in security-critical systems) or the testing of the system may be completely outsourced. For such cases, being able to generate test cases from a separate test model brings flexibility to the testing process.

We used a test selection algorithm based on the satisfaction of assertions characterising the operations of the SuT. In order to make the satisfaction of these assertions more concrete, we introduced the *Alternative Scenarios Hypothesis*:

> For any operation of the SuT, if s satisfies the precondition but t does not, then their corresponding post-states must be different.

This hypothesis is relevant to robust systems, where we want to find situations where the Operation under Test (OuT) has incorrect fault-handling. In general, robustness testing checks that a system can handle unexpected user input or software failures by testing the software outside its expected input range. Thus, by feeding the software with classified unexpected input that fails one precondition at a time, the faults for different situations are uncovered. For the Tokeneer Experiment, a separate test model of the high integrity variant of the Tokeneer was produced in the Alloy modelling language [47]. The test case specifications were produced by falsifying the assertions of the operations modelled in some order. The test case specifications were then fed into Alloy Analyzer, and the test cases generated automatically as counterexamples using SAT-solving technology.

Bearing in mind the high quality of the system under test, including the fact that only five errors have previously been found in Tokeneer since its release, we wish to report a small success in our experiment. We detected nine anomalous behaviours, and we describe four representative scenarios in the rest of this section (see [3] for a more detailed account of how we found the anomalies and for a description of the other scenarios). In Section 17.6, we consider whether these anomalous scenarios really compromise the security of the enclave and its workstations.

17.4.1 Scenario 1

Our first scenario concerns tailgating. The Tokeneer system should be seen as more than the sum of its software and hardware peripherals. It is a *socio-technical* system, and there are interactions between people and their procedures and the hardware and software, and these have to be considered to get a picture of the entire system. We understand that users must undertake not to "tailgate," that is, to follow an authorised user into the enclave without being separately authorised themselves. So we can rule out the possibility of deliberate tailgating, since users agree not to do it, and no one has any malicious intent. But we found a scenario that can be explained by accidental tailgating. Consider the following sequence of events.

1. Miss Moneypenny, an NSA employee, inserts her smartcard into the reader, which checks its validity.
2. She places her thumb on the fingerprint reader, which checks her identity.
3. The system authorises Miss Moneypenny, unlocking the door.
4. Miss Moneypenny enters the enclave, and the door closes behind her.
5. I am a new NSA employee. Since I have not started to work on a project that needs access to the enclave area, my card does not have access to the enclave, but I am not told which areas my card has access to. I assume that my card has access to all areas.
6. I have eyes only for Miss Moneypenny, and as a result I look anxiously at the door to get a glimpse of her.
7. I insert my card into the reader.
8. The screen says

   ```
   ENTRY DENIED
   ```

   but I am not paying attention. There is no provision for audible alarms.
9. I place my thumb on the fingerprint reader, but again I miss the error message.
10. Anyone watching my actions would be satisfied that I appear to be following authorised procedures, and so would have no reason to be suspicious.
11. The door is still unlocked following Miss Moneypenny's entry, and it stays unlocked for a period known as the *latch-unlock duration*. I enter the enclave during this period, and there is no way of detecting my unauthorised entry.

The scenario may be thought rather far-fetched and scarcely credible. It relies on two mistakes: me not noticing either of the error messages. But it does show how I could accidentally gain access to the enclave, even though this appears to be an unlikely occurrence. Perhaps more interestingly, it reveals something about the implementation of the system.

1. Audible alarms could profitably be used to draw attention to the authorisation failures of the card and fingerprint readers.
2. The no-tailgating rule could be enforced with hardware (a physical turnstile or a pair of "airlock" doors), or checked with a video entry-detection system.

3. There is a vulnerability offered by the door being unlocked during the latch-unlock duration. There are two issues here.
   (a) The duration must be long enough to allow Miss Moneypenny to enter, but not so long as to give others the possibility to tailgate; this seems impossible to get right.
   (b) Errors triggered during attempted authorisations do not prematurely end the latch-unlock duration. This is a missed opportunity.

17.4.2 Scenario 2

Suppose now that the security officer decides to shorten the latch-unlock duration, perhaps as a result of discovering my antics in Scenario 1. The security officer needs to update a configuration file on one of the workstations within the enclave, and suppose that he wants to decrease the duration from 30s to 15s.

1. The security officer modifies the configuration file, prepares the configuration data, and writes it to a floppy disk.
2. The security officer successfully authorises his entry and then enters the enclave.
3. He successfully authorises his use of a workstation.
4. He logs in, inserts the floppy disk, and enters the `update` command, but then he sees the following:

   ```
   read/write error
   ```

5. The security officer is uncertain what this means, so he then checks the screen showing the new configuration file. It says very clearly:

   ```
   Latch-Unlock Duration = 15s
   ```

   and he is satisfied that the update really has taken place.
6. Working hours currently stop at 17:00, and entry to the enclave should be forbidden after this time.
7. But I accidentally misread the time, thinking it to be 16:45, when it is really 17:45. This is probably due to my having returned from a foreign trip, and having made a mistake adjusting between time zones.
8. I successfully authorise my entry at 17:45 and enter the enclave.
9. I work until 18:00, 1 h later than permitted.

This scenario is rather puzzling at first sight: how could I have been authorised to enter the enclave in Step 8, clearly outside working hours? To answer this, we need to know a little bit more about how Tokeneer works. There is a default configuration file on the system that is used in case of an update failure. When the system detects a `read/write error` while trying to read the floppy disk or when the file on the floppy disk is incorrectly formatted, it is forced to use this default configuration file. The default file is inaccessible, presumably to prevent accidental interference. It just

so happens in our scenario that the default value for the latch-unlock duration is the same as that required by the security officer (15s), and this coincidence persuades him in Step 5 that the update has taken place correctly. But of course, all the other settings now assume their default values. In our scenario, this gives the later working time of 18:00, and so explains my entry in Step 8.

So I have gained accidental entry to the enclave and accidental access to a workstation. Is this an unlikely scenario? It depends on how carefully the configuration data file is prepared and whether or not the floppy disk drive is working properly, and it may well be an ageing device with reliability problems. But if a failure does occur, then it is made worse by a human–computer interface problem: the consequences of the error are not made clear to the security officer.

17.4.3 Scenario 3

The third scenario also concerns configuration data. The system logs information about workstation usage in an audit file, and an alarm is signalled when the file reaches the *minimum log-size* for sounding the alarm.

1. The number of users and tasks increases, with the audit log filling up rapidly.
2. The audit manager visits the enclave several times a day to archive the log.
3. The security officer agrees to raise the threshold to reduce the number of alarms.
4. He copies new configuration data to a floppy disk, authorises his entry and enters the enclave.
5. He logs in to a workstation and updates the system.
6. To check that the update has occurred properly, he logs out, logs back in again and checks the minimum audit-log size. It has the value he requires, so he logs out and leaves the enclave.
7. But the alarm is triggered once more by the old threshold.
8. He discovers later that the update seems to take effect only after the system is rebooted, following a public holiday.

The triggering of the alarm in Step 7 is puzzling. The security officer assured himself in Step 6 that the update had taken place satisfactorily, but later discovers that this doesn't seem to be the case. Again, we need to know more about how Tokeneer works. Although a new file is introduced to the system, and when requested, it can be viewed on the screen, the actual configuration data used for alerting stays the same till the next system start-up. The same thing also happens when a disk is inserted into the system drive for an admin operation. If the disk is not inserted at the start-up time, then the system does not recognise the disk.

This scenario affects the usability of the system, as too many alarms deny proper users the service they require from the system. It is a rather surprising behaviour, and may constitute an accidental denial-of-service attack.

17.4.4 Scenario 4

Our final scenario also concerns configuration files.

1. The security officer wants to change the closing time from 17:00 to 16:00.
2. He prepares a configuration file and writes it to a floppy disk.
3. But he accidentally forgets to erase an old configuration file.
4. He enters the enclave and updates system.
5. I enter the enclave at 16:30, half an hour after the new closing time.

Again the scenario is puzzling, and to explain it we need to know how Tokeneer deals with configuration files. The update function checks the validity of configuration files in the order it finds them on the floppy disk. As there was an older file, this is the one that is used. In our scenario, it was the older file that had a later closing time, creating the problem. Once more, I have gained accidental access to both the enclave and its workstations.

17.5 Analysis

All these stories are revealed by testing 12 operations. The operations realise their intended functionality when all the preconditions are satisfied, at least for one pre-state. However, it is only when the test cases exercise system behaviour beyond the anticipated operational envelope that the stories such as the ones given in the previous sections are uncovered. The next section discusses these scenarios in the context of the documentation produced by Praxis-HIS and SPRE during the development of the high-integrity variant of Tokeneer IDS.

To understand the relevance and validity of these stories, we must analyse them with respect to the documentation provided, which explains expected system behaviour and expected security properties. Section 17.5.1 provides information about the security model of the TIS, and evaluates the findings accordingly.

17.5.1 Comparison with Security Model and Requirements

This section compares the scenarios with the system requirements document [19,21] and the security model of the re-developed version of TIS [20]. In [20], the security model of the TIS is identified with the following six security properties.

1. If the latch is unlocked by the TIS, then the TIS must be in possession of either a User Token or an Admin Token. The User Token must have valid ID, Privilege, and I&A Certificates, and either have a valid Authorisation Certificate or have a template that allowed the TIS to successfully validate the user's fingerprint. Or, if the User Token does not meet this, the Admin Token must have a valid Authorisation Certificate, with role of *guard*.

2. If the latch is unlocked automatically by the TIS, then the current time must be *close* to being within the allowed entry period defined for the User requesting access. The term *close* needs to be defined, but is intended to allow a period of grace between checking that access is allowed and actually unlocking the latch. *Automatically* refers to the latch being unlocked by the system in response to a user token insertion, rather than being manually unlocked by the guard.
3. An alarm will be raised whenever the door/latch is *insecure*. *Insecure* is defined to mean the latch is locked, the door is open, and too much time has passed since the last explicit request to lock the latch.
4. No audit data is lost without an audit alarm being raised.
5. The presence of an audit record of one type (e.g. recording the unlocking of the latch) will always be preceded by certain other audit records (e.g., recording the successful checking of certificates, fingerprints, etc.). Such a property would need to be defined in detail, explaining the data relationship rules exactly for each case.
6. The configuration data will be changed, or information written to the floppy, only if there is an administrator logged on to the TIS.

The first and second properties do not prevent the TIS from situations such as the one given in Scenario 1. Here are some of the root causes of these stories:

- None of these properties imposes the condition that the owner of the card shall be the person entering the enclave.
- There is no way of checking whether the person who is authorised outside the enclave actually enters the enclave after removing his/her card from the card reader.
- It is difficult to keep track of users using the enclave without the exit point, which was included in the actual Tokeneer system specification, but excluded in the redeveloped version of TIS.
- No action is taken after an access denial, even if the door is open.

One might argue that some of these statements are not set as the requirements of the high-integrity variant of the TIS, and some of them are even explicitly excluded in the documentation. However, this does not affect the validity of the stories mentioned above. Therefore, if any of them is taken under investigation in the future, one of the root causes listed above must be considered whilst looking for a solution.

Another discussion item relevant to one of the stories is Security Property 6. The property states that the configuration data will be changed by an administrator (more specifically by a Security Officer), and the information (log files) will be written to the floppy by an administrator (more specifically by the Audit Manager). As shown in Scenario 2, in the case of an invalid configuration file, the system replaces the current configuration file with a default one. In other words, if the security officer attempts to update the configuration file with an invalid file by mistake, it is the *system* that updates the file with some default file without the control of the security officer. By taking such an action without the approval of the security officer, the system actually breaks one of the security properties stated earlier. In the Security Properties Document of the TIS [19], it is declared that the proof of this security property is missed, therefore we will not discuss this item any further. However, it

is open to discussion whether more attention should be given to the correctness of a file that determines how long the door to a secure enclave can be kept open, the latch of this door can be kept unlocked, the enclave is available to users, etc.

Regarding system faults and the alarms raised, in [5] it is stated that the system faults are warnings, with the exception of critical faults listed as failure to control the latch, failure to monitor the door and failure to write to the Audit Log. It is also mentioned as a requirement that the system shall continue to function following a system fault categorised as a warning, and raise an alarm following a critical fault. During the test case execution process, it was not possible to concretise the test cases that require a critical fault, therefore we are not in a position to state the behaviour of the system under these circumstances. However, there were test cases that required the alarm to be raised due to other causes such as the door being kept open more than allowed or the audit file size exceeding the limit specified in the configuration file. The system continues to perform normally for such cases even though an alarm is raised. Our concern is that the system may behave similarly for the critical faults mentioned above since the only requirement specified following a critical fault is to raise an alarm. The security of the system may not be compromised if the log file is too large, but a failure to control the latch would certainly create a risk, and therefore we believe that the measures taken for the latter case should be more than just raising an alarm.

The next section explains the system-level testing carried out by SPRE Inc previously, and compares the test results of this testing activity with that explained in this report.

17.6 Conclusions

Is Tokeneer really secure? Of course it is! It seems very unlikely that any of the accidents described by our scenarios could really happen and compromise the security of the enclave and its workstations, particularly as an administrator is needed for three of the four scenarios. But these are interesting scenarios nonetheless, and they have been overlooked both by the formal development and by system testing. In fact, the system testing performed by SPRE detected Scenario 2, and the case is closed as solved; however, in our tests, it persisted, using tests generated by a completely different technique.

Additionally, these scenarios may be useful in designing similar systems in the future, as they raise questions about configuration files, audit logs, and alarms that may have been overlooked this time. Perhaps they might be useful if the system were to evolve to include stronger security guarantees, including malicious intent.

The lesson of our experiment is that there is value in diversity: an alternative approach gives us an opportunity to think laterally. De Bono's *lateral thinking* is about judging the correctness of a statement (in this case the correctness of Tokeneer), and seeking errors that contradict that claim of correctness. Even when a considerable amount of effort has been invested in one approach, in this case the

application of formal methods, lateral thinking, provided by using a completely different, but principled set of techniques and tools, can still challenge the claim of correctness.

Acknowledgements First, thanks must go to all those involved in the Tokeneer project for releasing into the public domain such a useful project archive. It is incredibly valuable and has done the research community a very great service. The work on Tokeneer reported in this paper was carried out as part of Emine Gökçe Aydal's Ph.D. thesis [3], under the supervision of Jim Woodcock. We also received helpful comments during the development of this work from Andrew Butterfield, Behzad Bordbar, Néstor Cataño, Ana Cavalcanti, John Clarke, John Fitzgerald, Leo Freitas, Rob Hierons, Tony Hoare, Randolph Johnson, Cliff Jones, Bertrand Meyer, Yannick Moy, Marcel Oliveira, Richard Paige, Brian Randell, Shankar and Angela Wallenberg. We presented the results of our experiment to Marie-Claude Gaudel's research group during a sabbatical visit to Université de Paris-Sud, and to audiences in seminars and workshops at the University of Birmingham, Trinity College Dublin, the University of Madeira, Microsoft Research Asia, Microsoft Research Cambridge, the Federal University of Rio Grande do Norte, the University of York and ETH Zurich. We are grateful for all the encouragement we received.

References

1. Aichernig B.K., Maibaum, T.S.E. (eds.): Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18–20, 2002, Revised Papers, volume 2757 of Lecture Notes in Computer Science. Springer (2003).
2. Aydal, E.G., Paige, R.F., Woodcock, J.: Evaluation of OCL for large-scale modelling: A different view of the Mondex purse. In: Giese, H. (ed.) MoDELS Workshops, volume 5002 of Lecture Notes in Computer Science, pp. 194–205. Springer, Berlin, Heidelberg (2007).
3. Aydal, E.G.: Model-Based Robustness Testing of Black-box Systems. PhD thesis, Department of Computer Science, University of York (November 2009).
4. Banach, R.: Formal Methods: Guest editorial. J. UCS, **13**(5), 593–601 (2007).
5. Barnes, J.: Tokeneer ID Station informed design. Technical Report S.P1229.50.2, Praxis High Integrity Systems. Available from tinyurl.com/tokeneer (2008)
6. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the Tokeneer enclave protection system. In Proceedings of the 1st International Symposium on Secure Software Engineering, Arlington, VA. IEEE (March 2006).
7. Barnes, J.: High Integrity Ada: The SPARK Approach to Safety and Security. Addison-Wesley, Reading, MA. (2003).
8. Bicarregui, J., Fitzgerald, J.S., Larsen, P.G., Woodcock, J.C.P.: Industrial practice in Formal Methods: A review. In: A. Cavalcanti and D. Dams (eds.) FM, volume 5850 of Lecture Notes in Computer Science, pp. 810–813. Springer (2009).
9. Bicarregui, J., Hoare, C.A.R., Woodcock, J.C.P.: The Verified Software Repository: A step towards the verifying compiler. Formal Asp. Comput. **18**(2), 143–151 (2006).
10. Butler, M., Yadav, D.: An incremental development of the Mondex system in Event-B. Formal Asp. Comput. **20**(1), 61–77 (2008).
11. Butterfield, A., Freitas, L., Woodcock, J.: Mechanising a formal model of flash memory. Sci. Comput. Program. **74**(4), 219–237 (2009).
12. Butterfield, A. O'Cathain, A.: Concurrent models of flash memory device behaviour. In M. Oliveira and J. Woodcock, editors, Brazilian Symposium on Formal Methods (SBMF 2009), 19–21 August 2009, Gramado, Brazil, Lecture Notes in Computer Science, in press. Springer (2009).

13. Butterfield, A., Woodcock. J.: Formalising flash memory: First steps. In 12th International Conference on Engineering of Complex Computer Systems (ICECCS 2007), 10–14 July 2007, Auckland, New Zealand, pp. 251–260. IEEE Computer Society (2007).
14. CCRA.: Common criteria for information technology security evaluation. Part 1: Introduction and general model. Technical Report CCMB-2006-09-001, Version 3.1, Revision 1, Common Criteria Recognition Agreement September (2006).
15. Chapman, R.: Tokeneer ID Station overview and reader's guide. Technical Report S.P1229.81.8, Issue 1.0, Praxis High Integrity Systems. Available from tinyurl.com/tokeneer (2008)
16. Chapman, R.: Private communication. Email, 16 December 2009.
17. Cohen, E.: Validating the Microsoft Hypervisor. In: Misra, JV., Nipkow, T., Sekerinski, E. (eds.) FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21–27, 2006, Proceedings, volume 4085 of Lecture Notes in Computer Science, pp. 81–81. Springer (2006).
18. Cooke, J.: Editorial (VSTTE special issue). Formal Asp. Comput. **19**(2), 137–138 (2007).
19. Cooper, D., Tokeneer ID Station security properties. Technical Report S.P1229.40.4, Praxis High Integrity Systems. Available from tinyurl.com/tokeneer (2008)
20. Cooper, D.: Tokeneer ID Station security target. Technical Report S.P1229.40.1, Praxis High Integrity Systems. Available from tinyurl.com/tokeneer (2008)
21. Cooper, D.: Tokeneer ID Station system requirements specification. Technical Report S.P1229.41.1, Praxis High Integrity Systems. Available from tinyurl.com/tokeneer (2008)
22. Craig, I.D.: Formal Models of Operating System Kernels. Springer (2006).
23. Craig, I.D.: Formal Refinement For Operating System Kernels. Springer (2007).
24. Crocker, D., Carlton, J.: Verification of C programs using automated reasoning. In Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007), 10–14 September 2007, London, England, UK, pages 7–14. IEEE Computer Society (2007).
25. Damchoom, K., Butler, M.: Applying event and machine decomposition to a flash-based filestore in Event-B. In: Oliveira, M., Woodcock, J. (eds,) Brazilian Symposium on Formal Methods (SBMF 2009), 19–21 August 2009, Gramado, Brazil, Lecture Notes in Computer Science, in press. Springer (2009).
26. Damchoom, K., Butler, MJ., Abrial, J-R.: Modelling and proof of a tree-structured file system in Event-B and Rodin. In Liu, S., Maibaum, T.S.E.. Araki, K. (eds,) Formal Methods and Software Engineering, 10th International Conference on Formal Engineering Methods, ICFEM 2008, Kitakyushu-City, Japan, October 27–31, 2008. Proceedings, volume 5256 of Lecture Notes in Computer Science, 25–44. Springer (2008).
27. Deharbe, D: Modelling FreeRTOS with B. In Oliveira, M., Woodcock, J. (eds,) Brazilian Symposium on Formal Methods (SBMF 2009), 19–21 August 2009, Gramado, Brazil, Lecture Notes in Computer Science, in press. Springer (2009).
28. Dong, J.S., Sun, J.: SCP special issue on the Grand Challenge—Preface. Sci. Comput. Program. **74**(4), 167 (2009).
29. Ferreira, M.A., Oliveira J.N.: Towards tool integration and interoperability in the GC: The Intel flash file store case study. In Marcel Oliveira and Jim Woodcock, editors, Brazilian Symposium on Formal Methods (SBMF 2009), 19–21 August 2009, Gramado, Brazil, Lecture Notes in Computer Science, in press. Springer (2009).
30. Freitas, L.: Mechanising data-types for kernel design in Z. In: Oliveira M., Woodcock, J. (eds,) Brazilian Symposium on Formal Methods (SBMF 2009), 19–21 August 2009, Gramado, Brazil, Lecture Notes in Computer Science, in press. Springer (2009).
31. Freitas, L., Fu, Z., Woodcock, J.: POSIX file store in Z/Eves: an experiment in the Verified Software Repository. In 12th International Conference on Engineering of Complex Computer Systems (ICECCS 2007), 10–14 July 2007, Auckland, New Zealand, 3–14. IEEE Computer Society (2007).
32. Freitas, L., Woodcock, J.: Mechanising Mondex with Z/Eves. Formal Asp. Comput., **20**(1), 117–139 (2008).
33. Freitas, L., Woodcock, J., Fu, Z.: POSIX file store in Z/Eves: An experiment in the Verified Software Repository. Sci. Comput. Program., **74**(4), 238–257 (2009).

34. Freitas, L., Woodcock, J., Zhang, Y.: Verifying the CICS File Control API with Z/Eves: An experiment in the Verified Software Repository. Sci. Comput. Program., **74**(4), 197–218 (2009).
35. Gal, E., Toledo, S.: Algorithms and data structures for flash memories. ACM Comput. Surv., **37**(2), 138–163 (2005).
36. George, C, Haxthausen, A.E.: Specification, proof, and model checking of the Mondex electronic purse using RAISE. Formal Asp. Comput. **20**(1), 101–116 (2008).
37. Gomes, A O Oliveira, M.V.M: Formal specification of a Cardiac Pacing System. In A. Cavalcanti and D. Dams, editors, Formal Methods Symposium (FM 2009), 31 October–6 November 2009, Eindhoven, Lecture Notes in Computer Science, in press. Springer, (2009).
38. Graydon, P.J., Knight, J.C., Strunk, E.A.: Achieving dependable systems by synergistic development of architectures and assurance cases. In R. de Lemos, C. Gacek, and A. B. Romanovsky, editors, WADS, volume 4615 of Lecture Notes in Computer Science, pp. 362–382. Springer (2006).
39. Haneberg, D., Schellhorn, G., Grandy, H., Reif, W.: Verification of Mondex electronic purses with KIV: from transactions to a security protocol. Formal Asp. Comput., **20**(1), 41–59 (2008).
40. Hoare, C.A.R.: Towards the verifying compiler. In Aichernig and Maibaum [1], pp. 151–160 (2003).
41. Hoare C.A.R., Misra, J.: Preface to special issue on software verification. ACM Comput. Surv. **41**(4) (2009).
42. Hoare, C.A.R., Misra, J., Leavens, G T., Shankar, N.: The verified software initiative: a manifesto. ACM Comput. Surv. **41**(4) (2009).
43. Hoare, T., Atkinson, M., Bundy, A., Crowcroft, J., Crowcroft, J., Milner, R., Moore, J., Rodden, T., Thomas, M.: The Grand Challenges Exercise of the UKCRC. report to the UKCRC from the programme committee. tiny.cc/gcreport (29 May 2003).
44. Hoare, T., Jones, C., Randell, B.: Extending the horizons of DSE. In Grand Challenges. UKCRC, 2004. tinyurl.com/ExtendingDSE (2004)
45. Hoare, T., Misra, J.: Verified software: Theories, tools, and experiments: Vision of a Grand Challenge project. In: Meyer, B. and Woodcock, J. (eds,) Verified Software: Theories, Tools, and Experiments. First IFIP TC2/EG2.3 Conference, Zurich, October 2005, volume 4171 of Lecture Notes in Computer Science, pp. 1–18. Springer, Berlin, Heidelberg (2008).
46. ITSEC. Information technology security evaluation criteria (ITSEC): Preliminary harmonised criteria. Technical Report Document COM(90) 314, Version 1.2, Commission of the European Communities June (1991).
47. Jackson, D.: Alloy: A lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (2002).
48. Jackson, D., Thomas, M., Millett, L.I., (eds.): Software for Dependable Systems: Sufficient Evidence? Committee on Certifiably Dependable Software Systems, National Research Council. The National Academies Press (2007).
49. Jackson, P., Passmore, G.O.: Improved automation for SPARK verification conditions. tiny.cc/jacksonspark (1 August 2009).
50. Jackson, P., Passmore, G.O.: YAFFS (Yet Another Flash File System). www.yaffs.net/ (1 August 2009).
51. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4) (2009).
52. Jones, C.B., O'Hearn, P.W., Woodcock, J.: Verified software: A Grand Challenge. IEEE Computer **39**(4), 93–95 (2006).
53. Jones, C.B., Pierce, K.G.: What can the π-calculus tell us about the Mondex purse system? In 12th International Conference on Engineering of Complex Computer Systems (ICECCS 2007), Auckland, New Zealand, 10–14 July 2007, pages 300–306. IEEE Computer Society (2007).
54. Jones, C.B., Woodcock, J.: Editorial. Formal Asp. Comput. **20**(1), 1–3 (2008).
55. Josey, A.: The Single UNIX Specification Version 3. Open Group, San Francisco, CA (2004.) ISBN: 193162447X.
56. Joshi, R., Holzmann, G.J.: A mini challenge: Build a verifiable filesystem. Formal Asp. Comput. 19(2), 269–272 (2007).

57. Kang, E., Jackson, D.: Formal modeling and analysis of a flash filesystem in Alloy. In E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors, ABZ2008: Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, September 16–18, 2008, volume 5238 of Lecture Notes in Computer Science, pp. 294–308. Springer, Berlin, Heidelberg (2008).
58. Kim, M.: Concolic testing of the multisector read operation for a flash memory. In M. Oliveira and J. Woodcock (eds.) Brazilian Symposium on Formal Methods (SBMF 2009), 19–21 August 2009, Gramado, Brazil, Lecture Notes in Computer Science, in press. Springer (2009).
59. King., J.C.: A Program Verifier. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, (1969).
60. Kuhlmann, M., Gogolla, M.: Modeling and validating Mondex scenarios described in UML and OCL with USE. Formal Asp. Comput. **20**(1), 79–100 (2008).
61. Lawford, M.: Pacemaker Formal Methods Challenge. tiny.cc/pacemaker, 1 (August 2009).
62. Macedo, H.D., Larsen, P.G., Fitzgerald, J.S.: Incremental development of a distributed real-time model of a cardiac pacing system using VDM. In: Cuéllar, J., Maibaum, T. and Sere, K. (eds.) FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26–30, 2008, Proceedings, volume 5014 of Lecture Notes in Computer Science, pp. 181–197. Springer, (2008).
63. Machado, P.: Automatic test case generation of embedded real-time systems with interruptions for FreeRTOS. In: Oliveira, M. and Woodcock, J. (eds.) Brazilian Symposium on Formal Methods (SBMF 2009), 19–21 August 2009, Gramado, Brazil, Lecture Notes in Computer Science, in press. Springer, (2009).
64. Meyer, B., Woodcock, J., (eds.): Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10–13, 2005, Revised Selected Papers and Discussions, volume 4171 of Lecture Notes in Computer Science. Springer (2008).
65. Mühlberg, J.T., Lüttgen, G.: Verifying compiled file system code. In M. Oliveira and J. Woodcock, editors, Brazilian Symposium on Formal Methods (SBMF 2009), 19–21 August 2009, Gramado, Brazil, Lecture Notes in Computer Science, in press. Springer (2009).
66. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, Juny 15–18 1992, volume 607 of Lecture Notes in Computer Science, pages 748–752, Springer, Berlin, Heidelberg (1992).
67. Ramananandro, T.: Mondex, an electronic purse: Specification and refinement checks with the Alloy model-finding method. Formal Asp. Comput. **20**(1), 21–39 (2008).
68. Shankar, N.: Automated deduction for verification. ACM Comput. Surv. **41**(4) (2009).
69. Shankar, N., Woodcock, J., (eds.): Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6–9, 2008. Proceedings, volume 5295 of Lecture Notes in Computer Science. Springer (2008).
70. Spinellis, D.: A look at zero-defect code. tinyurl.com/spinellisblog (18 October 2008).
71. Spivey, J.M.: The Z Notation: a Reference Manual. International Series in Computer Science. Prentice Hall (1989).
72. Stepney, S., Cooper, D., Woodcock, J.: More powerful Z data refinement: Pushing the state of the art in industrial refinement. In: Bowen, J. P., Fett, A. and Hinchey, M. G. (eds.) ZUM, volume 1493 of Lecture Notes in Computer Science, pp. 284–307. Springer (1998).
73. Stepney, S., Cooper, D., Woodcock, J.: An electronic purse: Specification, refinement, and proof. Technical Monograph PRG-126, Oxford University Computing Laboratory July (2000).
74. Tokeneer, tinyurl.com/tokeneer (2009).
75. VSR.: Verified Software Repository. vsr.sourceforge.net/fmsurvey.htm (2009).
76. Woodcock, J.: E6: Use of formality, Video Tape G3A, Tape No. 68. Technical report, Government Communications Headquarters, Communications-Electronics Security Group (October 1997).
77. Woodcock, J.: First steps in the Verified Software Grand Challenge. IEEE Computer **39**(10), 57–64 (2006).
78. Woodcock, J., Banach, R.: The Verification Grand Challenge. J. UCS **13**(5), 661–668 (2007).

79. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. International Series in Computer Science. Prentice Hall (1996).
80. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.S.: Formal Methods: Practice and experience. ACM Comput. Surv. **41**(4) (2009).
81. Woodcock, J., Stepney, S., Cooper, D., Clark, J A., Jacob, J.: The certification of the Mondex electronic purse to ITSEC Level E6. Formal Asp. Comput. **20**(1), 5–19 (2008).