The Value of Verification: Positive Experience of Industrial Proof

Steve King[1], Jonathan Hammond[2], Rod Chapman[2], and Andy Pryor[2]

[1] Department of Computer Science, University of York,
Heslington, York, YO10 5DD, UK
king@cs.york.ac.uk
[2] Praxis Critical Systems, 20 Manvers St, Bath, BA1 1PX, UK
{jarh,rod,aap}@praxis-cs.co.uk

Abstract. This paper describes the use of formal development methods on an industrial safety-critical application. The Z notation was used for documenting the system specification and part of the design, and the SPARK subset of Ada was used for coding. However, perhaps the most distinctive nature of the project lies in the amount of proof which was carried out: proofs were carried out both at the Z level — approximately 150 proofs in 500 pages — and at the SPARK code level — approximately 9000 verification conditions generated and discharged. The project was carried out under UK Interim Defence Standards 00-55 and 00-56, which require the use of formal methods on safety-critical applications. It is believed to be the first to be completed against the rigorous demands of the 1991 version of these standards.
The paper includes a comparison of proof with the various types of testing employed, in terms of their efficiency at finding faults. The most striking result is that the Z proof was substantially more efficient at finding faults than the most efficient testing phase. Given the importance of early fault detection, this helps to demonstrate the significant benefit and practicality of large-scale proof on projects of this kind.

1 Introduction

When early drafts of the UK Defence Standard 00-55[1] were produced, there was a certain amount of controversy among software suppliers because of the perceived emphasis on formal methods: a formal specification and design were required, as well as formal arguments to link the specification to the design, and the design to the code, and even to support the production of an executable prototype. It was claimed that the level of formality required was unrealistic given current technology. The work reported in this paper shows that developing software using such formal techniques is indeed possible. It is now becoming more common for projects to use formal notations to document specifications and even designs, but this project is unusual in the scale of the proof work that has been carried out. The particular notations used were Z [24, 28] for specification and design, and

[1] The procurement of Safety Critical Software in Defence Equipment [18, 19]

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. II, LNCS 1709, pp. 1527–1545, 1999.

the SPARK [3, 23] subset of Ada for code, together with its associated toolset. The proof work on Z covered about 500 pages, while over 9,000 verification conditions were generated and discharged in the SPARK proof work.

During the project, many metrics were recorded, and a selection are reported in this paper. It is interesting to compare the numbers of faults found at various stages of the process with the amount of effort spent on the stage. These figures seem to show the value of the proofs that were carried out on the Z documents.

The structure of the paper is as follows: after a brief description of the application, the SPARK programming language and toolset are described. (It is assumed that the reader is familiar with the Z notation: if not, there are several good text books available [28, 16], and a glossary is provided in Appendix A for a few key Z terms.) We then describe in some detail how proof was used in the development process, and look at both quantitative and subjective results before drawing some conclusions.

2 The Application: SHOLIS

The application we describe in this paper is called SHOLIS — the Ship Helicopter Operating Limits Information System. This is a new safety-critical system, which aids the safe operation of helicopters on naval vessels. It is essentially an information system, giving advice on the safety of helicopter flying operations. The SHOLIS programme is on-going but, after evaluation (if successful), it is intended that SHOLIS will be used on UK Royal Navy and Royal Fleet Auxiliary vessels. SHOLIS is developed for the UK Ministry of Defence (MoD) by PMES[2], with Praxis Critical Systems as the subcontractor responsible for the development of all the application software.

Brief System Description SHOLIS contains a database of Ship Helicopter Operating Limits (SHOLs). Each SHOL specifies the allowed limits for performing a given operation, e.g. takeoff or land, for a particular type of helicopter. One of the main safety-critical functions of SHOLIS is to make continual comparisons of sensor information against a selected SHOL. Audible and visual alarms are given whenever the current environmental conditions exceed the allowed limits.

The SHOLIS functions are grouped on a number of pages. These are viewed on plasma displays on the flight deck and bridge of a ship. Operators at each display can use buttons to view the pages independently. The buttons are also used to enter information, although certain functions, e.g. the selection of a SHOL, are only available to one display at any given time.

Due to its critical nature the system is developed to stringent standards, including the MoD Interim Defence Standards 00-55 [18, 19] and 00-56 [17], as discussed above. High availability requirements necessitate the use of dual redundant hardware.

[2] Power Magnetics and Electronic Systems Limited

Safety Requirements SHOLIS has a number of catastrophic hazards, which, if they occurred, could lead to the loss of an aircraft and/or damage to a ship. Any software with the potential to cause such a hazard, as identified by the software safety analysis, is classed as safety-critical and developed to SIL4[3]. The remaining software is classed as non-safety critical, although it is still developed to a stringent standard (roughly equivalent to SIL3).

3 The Programming Language: SPARK

SPARK is a high-level programming language, designed for writing software for high integrity systems. The executable part of the language is a subset of Ada [20], but there are additional annotations permitted which make it possible to carry out data and information flow analysis [4], and to prove partial code correctness, using the commercial toolset associated with the language: the SPARK Examiner, Simplifier and Proof Checker.[4]

There were several design drivers behind the choices as to what parts of Ada should be removed from the SPARK programming language:

- logical soundness: there should be no ambiguities in the language;
- simplicity of formal description: it should be possible to describe the whole language in a relatively simple way;
- expressive power: notwithstanding the previous two factors, the language should be rich enough to describe real systems;
- security: it should be possible to determine statically whether a program conforms to the language rules;
- verifiability: program verification should be not only theoretically possible, but also tractable for industrial-sized systems;
- bounded time and space requirements: in order to avoid the possibility of run-time errors caused by exhausting finite resources such as time and space, the resource requirements of a program should be determinable statically.

Together, these considerations led to decisions to omit several features of Ada: `goto`s, aliasing, default parameters for subprograms (i.e. procedures and functions), side-effects in functions, recursion, tasks, exceptions and generics. In addition, several other features, such as the type model, are simplified: no access types (pointers), type aliasing, derived types or anonymous types. Apart from these exclusions and restrictions, the normal Ada package structure is used for programming, with its distinction between package interfaces (or specifications) and package bodies. Within a package, further structuring is possible using procedures and functions and it is at this level that we can see the first of the new annotations.

[3] Interim Defence Standard 00-56 defines four Safety Integrity Levels (SILs), of which SIL4 is the most critical, and SIL1 is the least critical.

[4] SPARK and its toolset were originally developed by Program Validation Limited (PVL), which was incorporated into Praxis Critical Systems in 1994.

Annotations are comments which are ignored by an Ada compiler, but processed by the SPARK tools. The first group of annotations is concerned with data and information flow analysis[5]:

- `--# global`
- `--# derives`
- `--# own`
- `--# inherit`

The `--# global` and `--# derives` annotations between them specify the information needed for data and information flow analysis of individual subprograms. Data-flow analysis involves checking that global variables and parameters are used in the expected way: imported variables can only be read from, exported variables can be written to, and variables that are both imported and exported can be read from and written to. There are also checks that variables are not being read before being initialised with a value, that values are not overwritten before being read, that all imported variables are actually used somewhere, and so on.

Information-flow analysis uses the `--# derives` annotation where, for each output variable, a list is given of the imported variables on which its final value depends. These dependencies are checked by making an analysis of the expressions assigned to variables in the subprogram body. Both data- and information-flow analyses are decidable, and entirely automated by the SPARK Examiner.

The `own` and `inherit` annotations are used for scoping and structuring. The `own` annotation is used to declare the existence of state variables inside a package: the values of these variables are preserved between calls of subprograms in the package. The `inherit` annotation makes visible the items from another package scope, e.g. it allows the annotations in a package to refer to the own variables of the inherited package.

The second group of annotations is used for code verification:

- `--# pre`
- `--# post`
- `--# assert`
- `--# return`

The `pre` and `post` annotations are found in the specification of a procedure, and are used for the traditional precondition and postcondition of the procedure — `pre` gives a predicate on the input parameters and initial state (imported) variables, while `post` relates input and output parameters and initial and final state (exported) variables. On non-looping programs, the SPARK Examiner produces proof obligations by 'hoisting' the postcondition through the procedure body and checking that the supplied precondition implies this transformed postcondition. For looping programs, the `assert` annotation is used to specify the loop invariant. The verification conditions (VCs) generated by the Examiner for looping programs check for *partial* correctness: separate arguments are needed

[5] All annotations are prefixed by `--#`, with `--` being the Ada comment prefix

to consider loop termination, if total correctness is required. Finally, the `return` annotation is used to define (explicitly or implicitly) the result of a function, thus allowing checking of functions to be carried out at a more abstract level.

The SPARK Examiner has a mode of operation where, in addition to the VCs generated by the flow analysis and proof annotations, it also generates VCs which, if discharged, would guarantee that the SPARK program could not raise any run-time exceptions. The design of the SPARK language itself ensures that the Ada exceptions `Tasking_Error` and `Program_Error` can never arise in a SPARK program. In addition, since SPARK is designed so that the space requirements can be computed statically, it is possible to guarantee that `Storage_Error` cannot be raised. The only remaining possible exception is `Constraint_Error`, and the restrictions on the SPARK language mean that this can only be caused by a division check, an index check, a range check or an overflow check. When invoked with the run-time check (RTC) option, the SPARK Examiner generates VCs for the first three of these checks, and the VCs for the overflow check can be generated by the RTC plus Overflow option.

There are two possible routes for discharging the VCs produced by the Examiner: the Simplifier and the Proof Checker. The Simplifier is an automatic tool which carries out routine simplification using a collection of rules. If a VC cannot be discharged by the Simplifier, then a developer can invoke the Proof Checker, which is an interactive assistant allowing exploration of the problem and (it is hoped) the construction of a proof.

4 Proof in the SHOLIS Development Process

4.1 The Development Process

The development process used for SHOLIS was a fairly standard one, following the requirements of IDS 00-55. In simplified form, it comprised:

- Requirements, written in English;
- Software Requirement Specification (SRS), written in Z and English;
- Software Design Specification (SDS), written in SPARK, Z and English;
- Code, written in SPARK;
- Testing.

The Requirements documents consisted of over 4,000 statements of system requirements, most of which were software-related, while the SRS was about 300 pages long, containing Z, English and some additional mathematical definitions (of vector geometry). The purpose of the SDS was to add implementation details to the SRS: software architecture, 'refinement' of one part of the Z specification (where an intermediate level of design was needed), scheduling design, resource usage, SPARK package specifications and so on. The software itself totalled about 133,000 lines of code, made up of 13,000 lines of Ada declarations, 14,000

lines of Ada statements, 54,000 lines of SPARK flow annotations,[6] 20,000 lines of SPARK proof annotations and 32,000 blank or comment lines.[7]

4.2 Proof Activities

The proof activities on SHOLIS can be divided into two areas: Z proof and SPARK proof. Proof of various Z properties took place at both SRS and SDS level. The SRS, containing the abstract Z specification, has several standard opportunities for proof: consistency of global variables and constants, existence of initial states and checking of preconditions. It is interesting to see how the structure of the Z specification was exploited in these proofs. The main structuring of the Z specification was by what were called 'subsystems', i.e. the state was partitioned into a number of pieces that were separately specified, together with 'local' operations. Subsystems included such things as sensors, alarms, faults, and the currently selected SHOL, etc. Each main display page selectable by the user also had its own subsystem. In general, the complete SHOLIS state was simply the conjunction of all the subsystems, with appropriate additional invariants.

The notable exception to this involved the pages, as SHOLIS has two displays. So a display state schema was defined (which included the various pages as schema types), and a standard functional promotion carried out to the complete multiple displays state. Thus an individual page's state schema is effectively promoted twice (once to the 'display level' and again to the 'multiple displays level').

Each of the 14 top-level (system) operations had a precondition proof. At the top level this consisted of manual rigorous argument to remove Ξ schemas (on unaffected subsystems) and associated top-level invariants. The rigorous arguments continued until the precondition proof had been 'factored down' to precondition proofs of the constituent subsystem operations. Sometimes top-level invariants (which were not obviously preserved) were also 'factored down', so the subsystem precondition proofs were sometimes stronger than the 'standard' Z precondition proofs (in that there was an additional 'factored down' invariant to preserve). In general, schema expansion was not performed until the subsystem level was reached. Proofs were only carried out for subsystems that had been identified as SIL4. This structuring did not apply to the initial state proof. Here the top-level obligation was mechanically fully expanded and then simplified.

Each subsystem was a separate chapter in the SRS and mapped to a different Ada package in the design.

[6] The SPARK flow annotations are deliberately written in a very 'spread-out' way which uses a large number of lines, for ease of maintenance. Furthermore the need to do code proof, coupled with the current lack of abstract proof support, meant that the SPARK concept of own variable refinement could not be exploited to reduce substantially the size of the annotations (see section 5.2).

[7] There was a little non-SPARK code: some assembler, used only in booting up SHOLIS, and some non-SPARK Ada, used for interfacing to hardware devices.

The key safety properties of SHOLIS were also formalised in Z, and proved. These properties were expressed in terms of a sequence of operations. Clearly, for this application, the most important safety properties involved ensuring that when certain sensor values were outside the current SHOL, a warning had to be given, and also that when the values were inside the SHOL, no alarm would be given.[8] Thus the proofs of safety properties involved checks of the form

$$In \fatsemi Calc \fatsemi Out \qquad \text{gives the correct warning} \quad ,$$

where the schema In verified and stored the input values, $Calc$ performed the comparison with the current SHOL and updated the alarm state, and Out gave the output processing. Each of these schemas represents one of the 14 top-level system operations, and thus the safety properties could not be expressed as part of the main specification, as they cover sequences of system operations.

At the SDS level there were further proof opportunities, demonstrating the consistency and correctness of the part of the design written in Z.

All of the proofs at the Z level were carried out by a form of 'rigorous argument', with some assistance from tools — particularly the CADiZ tool [26], for schema expansion. For the SPARK proof work, on the other hand, all of the work was carried out with machine assistance: the Examiner, Simplifier and Proof Checker were used.

Data- and information-flow analysis was carried out for all of the code in SHOLIS. The intention had been to do only data-flow analysis for the main control loop, and parts of the event scheduler called from this control loop. This is because, at that level, almost every variable has an effect on every other variable, so the information-flow annotations would be both lengthy and uninformative. However, the SHOLIS application software is a single process running on a single processor (modulo redundant hardware), containing software of different integrity levels. Hence, full information-flow analysis was needed at the top level to demonstrate functional separation between the SIL4 and non-SIL4 code, e.g. to show that the non-SIL4 code did not incorrectly interfere with critical data on the same processor.

The demonstration of functional separation also justified only constructing SPARK program correctness proofs on the SIL4 parts of the software. For every subprogram of this sort, SPARK `pre` and `post` annotations were produced from the Z descriptions. The SPARK names were kept as close as possible to the Z names, but there were inevitable small differences, for instance package names. There were also simple type translations: Z sequences became arrays with a slightly different syntax, partial functions also became arrays and so on. Although this could be seen as a 'weak link' in the formal development process, experience showed that it was actually relatively simple to produce these SPARK annotations, and very few detected errors were introduced at this point. The Z state invariants were incorporated into both `pre` and `post` annotations of procedures, which produced one or two interesting difficulties: the annotations could only refer to variables which were visible according to the SPARK rules,

[8] Otherwise safe recovery of aircraft might not be possible.

but sometimes the invariants referred to variables which were not visible. The solution was to write the strongest condition possible using the visible variables, so that, at the next level 'up', this condition together with the frame knowledge that other variables were unchanged would establish the invariant.

Although the intention had originally been to generate the proof annotations along with the code, time pressures — caused by the need to pass the code to the IV&V team[9] — meant that many proof annotations were actually added slightly later. Having produced the necessary annotations, the SPARK Examiner was then used to generate the proof obligations to show that the code did indeed satisfy its specification. These proof obligations were first submitted to the SPARK Simplifier, which managed to discharge about 75% of them automatically. The remaining ones were virtually all proved using the SPARK Proof Checker, the exceptions being:

- proof obligations that depended on formal descriptions of hardware devices which were not available; and
- proof obligations for a few subprograms, that involved a lot of effort to prove, but which, by symmetry, were merely further examples of code that had already been proved.

The final group of SPARK proof activities concerned the run-time checks (RTCs). Since it was clear that a run-time failure — be it invalid range or index, division by zero, or overflow — would be a danger to the safety-critical parts of SHOLIS whether it occurred in SIL4 code or not, the whole of the software was subjected to the SPARK Examiner's RTC (plus Overflow) facility. Again, all of the generated proof obligations were proved, either by the Simplifier or using the Proof Checker.

4.3 Proof Personnel

The proof activity on the SHOLIS project was carried out by four engineers. Two were responsible for the Z proofs, and one of these also worked with the other two engineers on the generation of SPARK proof annotations corresponding to the Z specifications, and all the SPARK proof activity. The data- and information-flow analysis was carried out by the two coders. All of the proof engineers were experienced mathematicians and software engineers who had worked for several years in various formal methods, including Z and CSP. However, only one had experience with the SPARK Simplifier and Proof Checker before the project started.

It is also interesting to consider, with hindsight, the skills which seem to be *necessary* for such a project. For the Z proof work, significant experience (either academic or industrial) of Z and at least some exposure to proof are necessary to be productive enough to be commercially cost-effective (e.g. familiarity with concepts such as proof by cases and proof by contradiction). For the *formal*

[9] Independent Verification and Validation team: part of Praxis Critical Systems, but independent of the development team.

SPARK proofs, a good (informal) understanding of the meaning of imperative programming constructs is essential, together with some familiarity with relevant proof concepts such as loop invariants. However, previous experience with the tools is not thought necessary. Interestingly, the proofs of absence of run-time errors are much more accessible, since the tools can generate the VCs without any additional proof annotations (although annotations may be needed to enable the VCs to be proved). Also, a large proportion of these VCs are typically proved automatically using the Simplifier. This enables effort to be quickly focussed on potential problem areas and/or the more complex code, where it may not be straightforward to prove the code error-free.

4.4 Proof Validation

The Z proofs were subject to a formal peer-review process, when the proofs produced by each engineer were formally reviewed by the other. In addition, the IV&V team reviewed a sample (selected by them) of the proofs, and found only typographical errors. The SPARK code proofs were also reviewed by the IV&V team, and are replayable on the SPARK toolset. The team also reviewed the additional proof rules that had been inserted to discharge the VCs.[10] However, none of the proofs was inspected or reviewed by the customer.

4.5 Timing and Resource Usage

As already discussed, the SHOLIS application consists of both SIL4 and non-SIL4 code. Although the information-flow analysis demonstrated functional separation, non-functional interactions (e.g. slow performance of non-SIL4 code preventing the timely execution of SIL4 code) could still have had an unacceptable impact on safety. So, in addition to functional correctness, significant effort was spent on non-functional aspects of the behaviour of *all* the SHOLIS code.

Timing: An in-house static timing analysis tool was used, which was based on programmer-supplied annotations in the source code. This did not read or analyse the object code at all — it merely computed a worst-case number of "statements" for each subprogram, and used a constant "number of statements per second" (determined by hand analysis and actual timing of a "typical" portion of the code) to make a crude estimate of an upper-bound on the timing of a subprogram.

Memory: Care was taken never to allocate memory dynamically: SPARK ensures this 99% of the time, but there were a few cases where careful coding was necessary to take into account the compiler's allocation policy.[11] SPARK is non-recursive, so a simple static analysis of object code is sufficient to determine worst-case stack usage, which was done.

[10] These rules were either the necessary definitions of SPARK proof functions, or more generally useful rules which are not part of the Proof Checker's rulebase.

[11] Section 5.2 contains more details on this topic.

I/O bandwidth: This was a crucial aspect of SHOLIS, since the available bandwidth to the displays was a limiting factor. Again, programmer-supplied annotations in the source (actually PERL expressions!) were used to indicate the worst-case number of characters that could be sent to the display by each subprogram. A simple PERL tool collected and evaluated the results.

The above systematic estimation/calculation was backed up in all cases with targetted testing, based on known worst-case application behaviour, to measure actual timing and resource usage (e.g. a dynamic "high water mark" test of stack usage). These tests provided additional confidence in the accuracy/conservative nature of the systematically produced figures.

5 Results, Experiences, and Lessons Learnt

Having described what was carried out in the way of proof on the SHOLIS project, we can now look at the results of this work, both in terms of quantitative results and in terms of more subjective feelings about the work.

5.1 Quantitative Results

In the Z proof work, approximately 150 proofs were carried out, of which about 130 were at the SRS level and the remainder at the SDS level. These proofs covered about 500 pages. In the SPARK proof work, approximately 9,000 verification conditions (VCs) were generated, of which 3,100 were proofs of functional and safety properties, and the remaining 5,900 came from the RTC generator. Of these 9,000 VCs, 6,800 were discharged automatically by the Simplifier and the remainder were discharged by the SPARK Proof Checker, or by the 'rigorous argument' referred to above, in a few cases. Indeed, subjective feedback from the project team emphasised the importance of using the most powerful workstations possible for the computationally intensive work of the Simplifier: 'a big computer is far cheaper than the time of the engineers using it'!

The project team kept track of faults found at different stages during the development process, and the rounded percentages are shown in Figure 1. The definition of a fault for these purposes is simply an error which required something to be changed. It could therefore range from a simple clerical error to an observable system failure. However, these figures exclude faults which were not faults in the actual system development (specification, design, code etc). Thus, for instance, errors in test scripts are not included. Figure 1 also shows how much of the total effort on SHOLIS (19 person-years) was spent on each phase.

Note that Figure 1 lists the project phases in approximately the order in which they occurred. However, there was some parallelism between phases. In particular, code proof overlapped with unit and integration testing, and especially with system validation testing.

For comparison, [13] contains figures on effort and size metrics for another safety-related real-time project, but for a much larger system than SHOLIS. This

Project phase	Faults found (%)	Effort (%)
Specification	3.5	5
Z proof	16	2.5
High-level design	1.5	2
Detailed design, code & informal test	26.5	17.5
Unit test	16	25
Integration test	1	1
Code proof	5.5	4
System validation test	21.5	9.5
Acceptance test	0.5	1.5
Other[12]	8	32

Fig. 1. Faults found and effort spent during phases of the project

other project also used formal methods, although there was only a very small amount of proof.

Informal feedback from the SHOLIS team indicated a feeling that the most cost-effective phases for fault-finding were Z Proof and System Validation Tests. The Z Proof phase in particular was felt to be effective at finding a significant number of faults, with relatively little effort, early in the development process. Figure 2 gives a graphical representation of the exact figures, where the dark bars show the actual number of faults found by each phase. For the verification phases, i.e. those phases whose main purpose was the detection of faults, Figure 2 also shows the efficiency with which faults were found (the lighter bars), by dividing the number of detected faults by the effort expended.

These figures clearly show that the Z Proof was, by a significant margin, the most efficient phase at finding faults, followed by the System Validation Test phase. It is perhaps even more surprising that Code Proof was more efficient than Unit Testing, despite the fact that substantial amounts of unit testing were completed before the bulk of code proof started.

One word of caution: it has not yet been possible to conduct a serious analysis of the nature of the faults (e.g. severity) found by different project phases. However, some initial impressions are described here.

The faults found during System Validation often originated from the requirements, or from incorrectly capturing the requirements in Z, rather than being instances of code not being a correct implementation of the Z specification. The faults found during the Code Proof phase were mostly cases of very subtle problems revealed by the RTCs — in particular circumstances (usually very unlikely ones), it might have been possible for a run-time error to have occurred. On the other hand, the traditional Unit and Integration Testing phases did find a number of faults that could have manifested themselves in realistic use of the final system. This included faults in two small, but critical, numerical calculations

[12] Staff familiarisation (1%), project management and planning (20%), safety management and engineering (7%) and IV&V non-testing activities (4%).

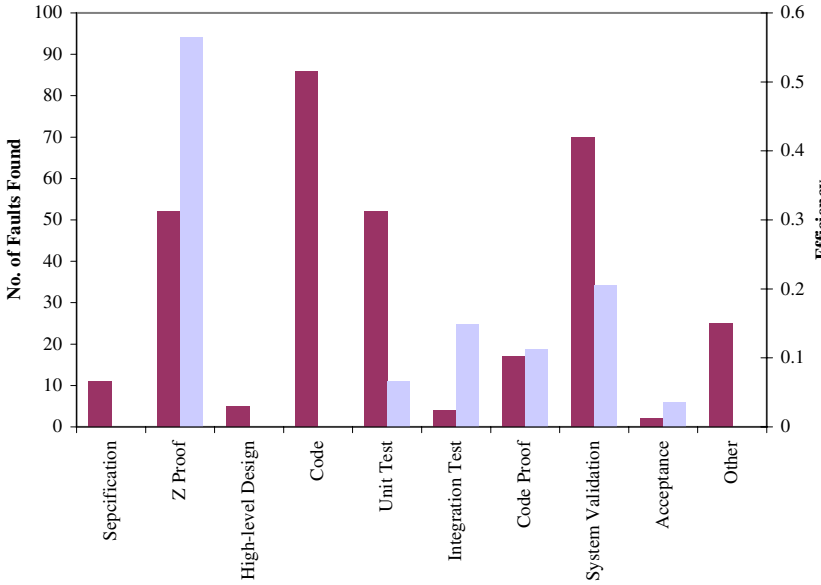

Fig. 2. Faults found and 'efficiency' of the phases of development

involving real (Ada fixed-point) arithmetic, where the SPARK proof model is not rich enough to allow precise reasoning about rounding and accuracy errors.[13] By sampling a few of the other faults found during unit testing, it has been confirmed that code proof should also have found the faults (since unprovable VCs are generated from the faulty code) if proof had occurred before testing. Even given the ordering of phases, the Code Proof phase did reveal one significant bug: the proof of the safety properties involved checking that certain invariants were maintained at the control loop level, but it was found that there was a path through the system which invalidated one of these invariants. Once this was understood, it was relatively easy for the developer to go to the test installation, press a few buttons and show that the system was in a clearly invalid state.

5.2 Subjective Feedback on the Use of Proof

Since this project was unique, in our experience, in the amount of Z/SPARK proof carried out, there were many lessons learnt, both about the advantages of doing these sorts of proofs, and about their drawbacks. One of the most important ideas to appreciate was the limit of formality. Figure 3 gives a representation of the call-tree of the main program: procedure Main is at the top of the tree, followed closely by the scheduler and event handler, while the subprograms and packages at the bottom include device drivers for the I/O devices.

[13] In these cases, manual numerical analysis was carried out to confirm the accuracy of the code.

Although the 'middle' part of the system could be neatly described by Z and SPARK, there were problems with both the 'top' and 'bottom' parts of the system. At the very top level, experience showed that the proof annotations were often simply too large to be manageable. This was exacerbated by the current lack of abstract proof support in SPARK, unlike the existing abstraction support for data and information flow analysis.[14]

Thus a decision was taken to prove only 'interesting properties' — such as the safety invariants — at the very top level of the SPARK. On the other hand, at the 'bottom' of the architecture, there was a need to interface with other software, such as device drivers, for which there was no formal specification at all. In this case, the solution adopted was often to supply a very abstract formal specification but no more. This usually took the form of a specification such as

$$o! = f(x) \quad ,$$

where the function f, acting on the state variables x to produce outputs $o!$, is left entirely nondeterministic. However, by naming f, it is possible to express the proof annotations, and to show exactly what properties of the supplied device driver are being relied on.

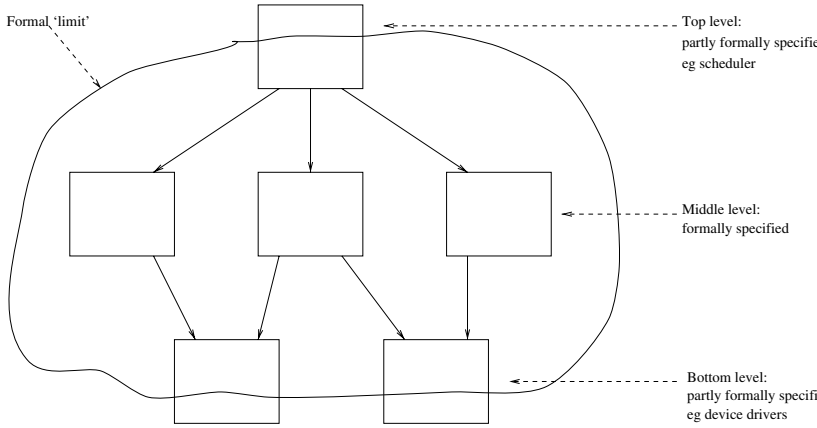

Fig. 3. The limits of formality

[14] SPARK includes the concept of an abstract own variable, where a single own variable (declared in a package specification) may represent a set of variables used in the implementation. Although the SPARK toolset fully supports abstract own variables in all types of flow analysis, there is currently no support for reasoning about such variable 'refinements'.

It was also important to remember that, of course, the development process did not stop when SPARK source code was produced: the code had to be compiled into object code. However good the development process had been in order to produce the source code, if the compiler had bugs, then the delivered system might be unacceptable due to compiler-introduced errors. A commercial, validated Ada compiler was used since that would bring some guarantees of quality through its years of service and the hope that any compiler bugs had been found by other users. In fact, the only compiler bug found during the project was in the optimiser, which was then switched off — the necessary performance was achieved by having the Ada run-time checks turned off in the compiled code. However, one difficulty with using a compiler for the full Ada language was that of course it did not understand the philosophy behind SPARK: on one occasion, it was realised that the compiler was using a perfectly valid code-generation strategy involving dynamic allocation of a large temporary variable, contrary to all of the SPARK ideas of predictability of resource usage, no dynamic memory allocation and so on. Here the solution involved a member of the project team using his ability to read the object code to write a script which checked the generated object code for dynamic memory allocation calls. There were also one or two problems in ensuring that the SPARK code was both provable and obeyed the timing requirements of the system. The fact that there was an expert in timing analysis [6] in the team was invaluable here.

In the Z proofs, it was found that the choice of state invariants was particularly important for finding errors: if the invariants were not strong enough, then it was quite possible to discharge the precondition proof obligation for an operation that had a postcondition which did not correspond to the desired outcome. (Some details of errors found by Z proofs are given below.)

There were several lessons learnt about coding styles which made the proof task easier: these are to be recorded in an internal 'Coding Style Guide' for future SPARK proof projects. For instance, if it is necessary to perform some action for every value of a small discrete type, it is sometimes easier to prove correct a sequence of statements rather than a loop over all of the possible values.

It is too early for there to be evidence yet about the cost of future changes to the system, though this is clearly an important question given the fairly novel and extensive use of proof on SHOLIS. Although there is limited experience of using tools to maintain proofs, it should be noted that the proportion of effort spent producing the proofs was fairly low (6.5%). One of the SHOLIS developers has remarked that he believes, as for most developments, that it is the design structure which is likely to have the most significant effect on the cost of future changes. There is also evidence [22], from an independent analysis of the project described in [13], that the use of a formal specification leads to simpler code which is easier to understand, and therefore to maintain.

Finally, at least one of the developers/provers remarked that it was from the VCs which *didn't* go straight through the Simplifier that most was learnt. In fact the code proof stage provided confidence that the code did actually implement

the Z specification, and the RTC proofs gave confidence that the Ada run-time check options on the compiler could be turned off safely.

5.3 The Types of Errors Found by Z Proofs

Approximately 70% of the Z proofs concerned preconditions, and they found approximately 75% of the total faults found by Z proof. An initial analysis of the faults reveals a number of different types (in approximately decreasing order of significance).

Incorrect functionality specified: there were several cases where, although the Z was well-defined and had the expected precondition, the actual functionality specified did not meet the requirements. These instances were found as a side-effect of the (human) prover having to understand precisely what the Z meant, in order to construct the proof, and realising that this did not correspond with their informal understanding of the required behaviour.

Lack of mode/history information modelled: as already described, SHOLIS has a number of different types of information pages. Certain pages and/or associated system functions are only available in particular circumstances, e.g. after a selection has been made. A number of precondition proofs revealed that the Z model did not adequately capture these ordering dependencies.[15] In each case, the solution was to add invariants to encode 'history' information, e.g. if this button (and hence system function) is available to the user then this selection state must be defined.

Contradictory operations: for a couple of operations, there were overlapping cases that specified conflicting behaviour, resulting in a contradiction. Perhaps more interesting were operations whose explicit postcondition predicates contradicted (implicitly included) invariants. In about four cases it turned out that the invariant was too strong, i.e. when the invariant was originally formulated, it was not noticed that there were legitimate situations where the invariant would not hold. Typically, these situations could be characterised and the invariant 'weakened' by the addition of an 'or'-case.

Missing cases: there were a number of instances of missing cases (e.g. not covering all possible combinations of input values). These typically resulted from either undefined function-applications (i.e. a value not being in a function's domain), or from the result of a calculation being outside an allowed range (e.g. to trying to increase a value beyond a fixed upper limit).

Incorrectly loose specifications: there were three or four examples where the prover spotted that the postcondition did not specify a value for one or more state components. Since Z has no 'rest unchanged' convention, for any variables which are to be left unchanged, this must be explicitly specified. As with the incorrect functionality case, it was not the precondition itself which showed the problem, but the thorough consideration of the operation required to produce the proof.

[15] Z has no explicit mechanism for specifying dependencies on the ordering of operations.

5.4 SPARK 83 versus SPARK 95

Since the SHOLIS project started in 1993, it was obviously not possible to use SPARK 95, the later version of SPARK derived from Ada 95, together with an updated toolset. However, it is clear that several features of SPARK 95 would have made life easier on the SHOLIS project: use type clauses, the ability to read out parameters, moded globals and the changes to static expressions. Some details of a later trial port of SHOLIS to SPARK 95 can be found in [7].

6 Related Work

While there has been an increasing use of formal methods for specification in industry — see, for example, [11, 12, 15] — there is less evidence for the use of refinement and proof. However, [25] offers a recent example of the use of Z refinement on an industrial scale. This work led to some improvements in the formulation of the Z refinement rules, and to a better understanding of the Z/CSP relationship [5].

On the SPARK proof front, [10] reports the use of SPARK with an extension to SPC's CoRE (Consortium Requirements Engineering) modelling method [8], which in turn is based on Parnas tables[1]. The specifications in these tables were converted to SPARK postconditions. Parnas tables were used successfully on the Darlington shutdown system [21, 9], but would not have been as appropriate as Z for the SHOLIS work, since Z has a much richer state-modelling capability. This was necessary for areas like maintaining a history of input sensor values.

7 Conclusions

The SHOLIS project made extensive use of formal methods, including both Z and SPARK proof, and it is believed to be the first to be completed under the 1991 version of UK MoD Interim Defence Standards 00-55 and 00-56.

The overall experience of industrial-scale proof has been very positive and obtained significantly better results than were originally expected. In terms of faults found for effort expended, the Z Proof phase was by far the most efficient phase of the project. One reason for this may be because the Z Proof was the first verification phase on the project. Proofs at the SPARK code level were not as efficient at finding faults, but this was to be expected since significant testing — both informal and formal — had already been completed before the code proofs took place. However, the code proofs were still more efficient at error detection than unit testing, and provided crucial assurance that the code was free of run-time exceptions.

The results of the different types of testing are also quite revealing. In particular, system validation testing was substantially more efficient at finding faults than unit testing. In our experience this is consistent with anecdotal evidence from other high-integrity projects. As a result we have significantly refined our testing strategy on more recent projects.

There are some important constraints to remember when attempting proof on a large-scale. Part of the success of proof on SHOLIS is due to the simple system architecture, and hence the straightforward mapping that is possible between the specification, design and code. If SHOLIS were a heavily distributed system, it is not believed that as much could have been achieved. (Further discussion of practical issues concerning the use of formal methods in large-system design can be found in [13, 14].) The limits of formality must also be considered. For the foreseeable future, testing is likely to have an important role in gaining necessary assurance of compilers, hardware, timing issues etc.

On the Z side, further support is needed (in terms of both proof techniques and tools) for reasoning about subsystems coupled by invariants, other than by brute force expansion. Some large-scale SPARK reasoning mechanisms are also needed, including some support for abstract proof, before the technology can be extensively used at the highest levels of large systems. The next release of the SPARK toolset is very likely to contain such mechanisms, as a result of experiences on SHOLIS and other projects.

In summary, proof was an important part of the SHOLIS development process, and an important factor in contributing to the quality of the delivered product. We believe our success shows both the significant benefit and practicality of large-scale proof on projects of this kind.

Acknowledgements

This paper reports the work of the entire SHOLIS proof and coding team: Janet Barnes, Rod Chapman, Jonathan Hammond, Andy Pryor and Neil White. The permission of PMES and MoD to publish this paper is gratefully acknowledged. The FM99 referees and Anthony Hall gave useful feedback on earlier versions of the paper.

A Z Glossary

This glossary gives brief definitions for the Z terms used in the paper. Readers are referred to the many text books on Z for a more extensive introduction.

Schema expansion: one of the key features of Z is the schema, a named collection of variable declarations and invariants linking them. The schema name can be used as a declaration, and this technique is widely used to control complexity. Schema expansion involves replacing schema names with the corresponding declarations and invariants. This can either be carried out 'all-in-one', when expansion continues until there are no schema names left, or 'one-level-at-a-time', when only the immediately-visible schema names are expanded — of course, this may introduce further schema names.

Ξ schema: a Ξ schema is used to describe operations which do not change the state of a system. It is a shorthand for the inclusion of a state before, a state after and an equality predicate stating that all state components are unchanged. It is typically used in 'enquiry' operations, where the purpose

of the operation is to give an output depending on the current state, rather than to change the state.

Promotion: this is a technique for specifying the behaviour of systems which consist of several copies of a smaller subsystem. The state of the subsystem is first described, together with operations on it. This 'local' state is then used in the description of the larger 'global' state, and the 'local' operations are combined with a *framing schema* to describe the operations on the global state. A 'functional promotion' is one where the local state is included in the global state by introducing a variable which is a function from an indexing set to the local state. Further details may be found in [28, 27, 2].

Precondition proof: in Z, operations are described with a single predicate, encapsulating both the precondition and the postcondition. The precondition can be extracted from this by applying the *pre* operator, which hides the after-state and outputs. The 'precondition proof' is then a check that the specifier's view of the operation's precondition — obtained by consideration of the environment in which the operation is executed — is strong enough to imply the real precondition, as expressed with *pre*.

Initial state proof: this is a proof that a valid initial state for the system does exist. In this context, 'valid' means 'obeying the state invariant'.

References

[1] T. Alspaugh, S. Faulk, K. Heninger Britton, R. Parker, D. Parnas, and J. Shore. Software requirements for the A7-E aircraft. Technical Report NRL/FR/5530-92-9194, Naval Research Laboratory, Washington, D.C., 1992.

[2] R. Barden, S. Stepney, and D. Cooper. *Z in practice*. BCS Practitioner Series. Prentice-Hall, 1994.

[3] J. Barnes. *High integrity Ada: The SPARK approach*. Addison-Wesley, 1997.

[4] J-F. Bergeretti and B.A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Prog. Lang. Sys.*, 7(1), January 1985.

[5] C. Bolton, J. Davies, and J.C.P. Woodcock. On the refinement and simulation of data types and processes. In K. Araki, A. Galloway, and K. Taguchi, editors, *IFM99: Proceedings of the 1st International Conference on Integrated Formal Methods*, pages 273–292. Springer-Verlag, 1999.

[6] R.C. Chapman, A. Burns, and A.J. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems Journal*, 11(2):145–171, September 1996.

[7] R.C. Chapman and R. Dewar. Re-engineering a safety-critical application using SPARK 95 and GNORT. In M.H. Harbour and J.A. de la Puente, editors, *Reliable Software Technology: Proceedings of the 1999 Ada Europe Conference, Santander, Spain*, number 1622 in Lecture Notes in Computer Science, pages 39–51. Springer-Verlag, 1999.

[8] Consortium Requirements Engineering Guidebook. Technical Report SPC-92060-CMC Version 01.00.09, Software Productivity Consortium, Herndon, VA, USA, 1993.

[9] D. Craigen, S. L. Gerhart, and T. J. Ralston. An international survey of industrial applications of formal methods. Technical Report NIST GCR 93/626-V1 & 2,

Atomic Energy Control Board of Canada, US National Institute of Standards and Technology, and US Naval Research Laboratories, 1993.

[10] M. Croxford and J.M. Sutton. Breaking through the V and V bottleneck. In M. Toussaint, editor, *Ada in Europe 1995*, volume 1031 of *Lecture Notes in Computer Science*, pages 344–354. Springer-Verlag, 1995.

[11] J. Fitzgerald, C. B. Jones, and P. Lucas, editors. *FME'97: Industrial Application and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*. Formal Methods Europe, Springer-Verlag, 1997.

[12] M.-C. Gaudel and J. C. P. Woodcock, editors. *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*. Formal Methods Europe, Springer-Verlag, 1996.

[13] A. Hall. Using formal methods to develop an ATC information system. *IEEE Software*, 13(2):66–76, March 1996.

[14] A. Hall. Keynote speech: What does industry need from formal specification techniques? In *2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques*, 1998.

[15] M.G. Hinchey and Bowen J.P., editors. *Applications of Formal Methods*. Prentice-Hall International series in computer science / C.A.R. Hoare, series editor. Prentice-Hall International, Englewood Cliffs, N.J. ; London, 1996.

[16] J. Jacky. *The way of Z: Practical programming with formal methods*. Cambridge University Press, Cambridge, UK, 1997.

[17] MOD. *Hazard analysis and safety classification of the computer and programmable electronic system elements of defence equipment*. UK Ministry of Defence, April 1991. INTERIM DEF STAN 00-56.

[18] MOD. *The procurement of safety critical software in defence equipment*. UK Ministry of Defence, April 1991. INTERIM DEF STAN 00-55 (Part 1: Requirements).

[19] MOD. *The procurement of safety critical software in defence equipment*. UK Ministry of Defence, April 1991. INTERIM DEF STAN 00-55 (Part 2: Guidance).

[20] K.A. Nyberg, editor. *The annotated Ada Reference Manual*. ANSI, 1983. ANSI/MIL-STD-1815A-1983.

[21] D.L. Parnas, G.J.K. Asmis, and J.D. Kendall. Reviewable development of safety critical software. In *Proceedings of the International Conference on Control and Instrumentation in Nuclear Installations*, 1990.

[22] S.L. Pfleeger and Hatton L. Investigating the influence of formal methods. *IEEE Computer*, 30(2):33–43, February 1997.

[23] *SPARK — The SPADE Ada Kernel*. Praxis Critical Systems, August 1997. Edition 3.3.

[24] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[25] S. Stepney, D. Cooper, and J.C.P. Woodcock. More powerful Z data refinement: Pushing the state of the art in industrial refinement. In J.P. Bowen, A. Fett, and M.G. Hinchey, editors, *ZUM'98: the Z formal specification notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 284–307. Springer-Verlag, 1998.

[26] I. Toyn and J.A. McDermid. CADiZ: An architecture for Z tools and its implementation. *Software — Practice and Experience*, 25(3):305–330, March 1995.

[27] J.C.P. Woodcock. Mathematics as a management tool: Proof rules for promotion. In B.A. Kitchenham, editor, *Software Engineering for Large Software Systems*. Elsevier, 1990.

[28] J.C.P. Woodcock and J. Davies. *Using Z: specification, refinement and proof*. Prentice-Hall International series in computer science / C.A.R. Hoare, series editor. Prentice Hall, 1996.