A Tale of Two Proofs

Susan Stepney

Logica UK Ltd. (stepneys@logica.com), and University of York

Abstract

One of the touted advantages of formal methods is the ability to do *proof*. But examples of proofs as part of industrial formal methods projects are relatively hard to find. I describe here two of the large Z proof projects I have been involved in at Logica, one for the correctness of a High Integrity Compiler, one for security properties of a smartcard-based electronic purse. I also show how the entire specification and proof process is deeply affected by *why* something is being proved, *what* is being proved, and how the finished proof is to be *presented*. I finish off by describing, based on my experiences, what I believe to be the requirements for an industrial-strength Z proof tool.

1. Introduction

Logica's Formal Methods Team has been involved in several large industrial-strength proof projects over the last few years. Two of these projects are the Z specification and correctness proof of a safety-critical compiler, and the Z specification and security proof of a smartcard-based electronic purse.

Formal specification 'keeps you honest', by making it that much more difficult to get away with imprecision and woolly thinking. But it is still possible to fool yourself with a specification that does not actually say what you think it does. Proof makes it yet harder to do this.

Proof is not an 'all-or-nothing' feature to be added to a project: proof can be performed at various levels of rigour, yielding various levels of benefits. As a consequence, proof does not provide any absolute guarantee of correctness; it provides an increased level of assurance in line with the degree of rigour used.

2. The two proof projects

2.1 Pasp-to-Asp Compiler, for AWE

In 1990 Logica's Formal Methods Team performed a study for RSRE (now DERA Malvern) into how to develop a compiler for high integrity applications that is itself of high integrity. We developed a technique for specifying a compiler and proving it correct, and built a small proof-of-concept prototype. The technique is described, and a small case study is worked up in full, including all the proofs, in [Stepney 1993].

The technique involves writing a formal specification of the compiler in terms of translation templates. The denotational semantics of the source and target language are also formally specified. The proof shows that the compiler is 'meaning preserving', that it always produces a target language program that has the same meaning as the corresponding source language program. The compiler specification is implemented in a high level language to give an executable compiler. (The source and target language specifications can also be implemented, to provide interpreters for these languages.)

Engineers at AWE read about the study and asked us to use these techniques to implement a compiler for their own high integrity processor, called the ASP (Arming System Processor). We have developed for them a high integrity compiler, integrated in a development and test environment, for a non-trivial subset of Pascal, including functions and procedures, and modules for separate compilation [Stepney 1998].

The specification of their source language Pasp, a Pascal-like subset, runs to about 150 pages. This includes specification of the concrete syntax and three static semantics (declaration before use, type correctness, and initialisation before use), as well as the dynamic semantics that is the starting point for the compiler correctness proofs. The Asp assembly language specification is rather shorter, at about 50 pages. The compiler specification – Asp templates corresponding to each Pasp construct – is about 100 pages, with a further 20 pages for the linker. The proof, which is presented mainly in outline, runs to about 50 pages.

The proof was performed chiefly to provide us, the implementers, with increased assurance that the development was correct, and also to allow the client to demonstrate that the development was of high integrity. Three highlights of the proof effort were:

1. **Correctness**: while casting the proof obligation for the function call expression, we realised that the evaluation stack as then specified would be overwritten when a function call was embedded in an arithmetic expression. This error was caught just by thinking about the proof obligation, not actually discharging it.
2. **Clarity**: when we came to add modules and linking to Pasp, we started by determining the proof obligations for separate compilation, and then used these to help us design the linking constructs themselves.
3. **Partial coverage**: The only flaw revealed in testing, beyond simple transcription errors, was that 16-bit unsigned division/modulus did not work for some arguments. But this was one of the few constructs that, due to the incremental nature of the project, we were still waiting to perform the proofs about. No errors were found in the constructs we had proved.

2.2 Electronic Purse, for NatWest Development Team

Over the past few years Logica's Formal Methods Team has been working with the NatWest Development Team proving the correctness of smartcard applications for electronic commerce.

The particular product I describe here is an 'electronic purse' hosted on a smartcard. Purses interact with each other, via a communication device, to exchange value. Once released into the field, each purse is on its own, and has to ensure the security of all its transactions without recourse to a central controller. All the security measures have to be implemented on the card, with no real-time external audit logging or monitoring.

This product is deeply security critical, and the client decided to use formal methods in their development process, in order to attain increased assurance. So we developed formal models of the implemented system and the abstract security policy, and proved that the system design possessed all the security properties [Stepney *et al.* 1998].

The specification of the abstract security policy is about 20 pages of Z. The concrete system specification, including the *n*-step value transfer protocol, is about 60 pages. The proof, presented at a level of detail suitable for external evaluation,

comprises 200 pages of refinement proof, and another 100 pages of derivation of the particular refinement rules we needed to use.

Highlights of this proof effort include:

1. **Understanding**: we made a key modelling discovery – of classifying certain partially completed transactions as 'definitely aborted' or 'maybe aborted' – which clarified our specification, and our understanding of the protocol. We would probably not have seen this without doing the proof.
2. **Backward rules**: we discovered that the traditional Z data refinement proof obligations [Spivey 1992, section 5.6] were not sufficient to prove our refinement. In particular, these obligations assume the use of a 'forward' (or 'downward') simulation, and allow no input/output refinement. But our models were linked by a 'backward' simulation, and needed i/o refinement. So we had to go back to first principles and derive the required rules.
3. **Clarification**: The amount of value in a purse part-way through the transfer protocol is not obvious. By having an abstract definition of the transfer, we were able to state precisely what the balance was, and so show that the protocol could not 'create' value, only transfer it.
4. **Proof discovered a bug**: our proof uncovered a bug in the proposed implementation of a secondary protocol. We were able to use the failed proof, plus the understanding that it gave us, to construct a scenario that was obviously flawed, and to fix the design so that it no longer had the flaw.
5. **Evaluation for yet more assurance**: the third-party evaluators found an unjustified assumption in one of our proofs; we were able to justify it fairly easily.

3. The proof process

3.1 Depth

We often hear words to the effect that "proofs in mathematics are deep; proofs in computer science are shallow", with the accompanying sub-text that shallow proofs are trivial and uninteresting.

It is certainly true that none of the individual proof steps in the two examples above required use of any particularly deep mathematics. Most of the steps were no more than applications of cut, one point, thin, Leibnitz, and Z toolkit laws. But just because each individual step is simple, does not mean that the whole proof is shallow. In our case, we have always needed to work out *why* we were proving something, just *what* we had to prove, and then *how* to prove it. And some of that meta-work was quite deep.

Agreed, the depth is not in the mathematical basis of the underlying formal specification language – typed set theory in the case of Z. The depth lies in the mathematics and context of the particular *model*. If that is deep, the proof can be deep and interesting, too.

3.2 Why prove

There are several different reasons one may want to do a proof.

- **Improve one's own understanding**: Just as for 'why *specify*?', proof can help one to communicate, to clarify, to understand, and to discover implicit properties of a specification.
- **Clarify real-world connections**: Sometimes the mapping between the specification and the real world is not what is thought. A clearly understood abstract concept might be refined to subtly different concrete concept, which is then implemented. (In the purse example, the concrete and abstract *values* are subtly different part way through the protocol. The security property is expressed in terms of the abstract value. Proof helped expose this difference.)
- **Increase one's own assurance**: Proof can increase one's confidence in something about the specification, such as the fact that a specification has some desired property, or that one specification is a refinement of another, or that two specifications both have the same property.
- **Third party assurance**: Proof can convince *others* that some property of the specification holds. The development is 'seen to be' correct. (For an interesting perspective on the role of proof, see [DeMillo *et al*. 1979].)
- **Certification/accreditation**: The most extreme achievement of third party assurance is to gain some form of badge of correctness, such as ITSEC E6.

The reason for doing a proof needs to be understood, because it can influence *what* to prove and *how* to prove it.

- **Intermediate steps**: A proof done to improve understanding needs to pass through a brain. So it should not be just mindless 'pushing the symbols around'. Intermediate stages in a proof should have an intuitive meaning of their own. (This can sometimes be difficult to achieve with a tool that stubbornly proceeds along its own route to *true*.)
- **Presentation**: whether a proof is for internal consumption or third party review affects the presentation style, and the amount of explanation.
- **Rigour**: If the proof is to pass formal evaluation criteria, it may be necessary to perform it to some minimum level of rigour.
- **Tools:** Whether to use a proof tool at all, and what kind of tool to use, again depends on the reason the proof is being done. If understanding needs to pass through someone's brain, then use a proof assistant, or do the proof by hand. If you need to know just that it is right, a more highly automated tool that says 'yes' or 'no' might be appropriate.

Our compiler proof was done mainly to help us know the development was correct, and to help us clarify the meaning of separate compilation. It had to pass through a brain, and so was done by hand. A secondary requirement was to give the client the ability to demonstrate that the development was correct to their own customers and quality department, which affected our style of presentation. The proof comprised a statement of the proof obligations, details for a few constructs, and sketches, outlines, and much use of 'similarly' arguments for the rest.

Our purse proof was done primarily to increase assurance, and also to be capable of achieving certification. This affected our presentation – the proof had to be understandable line-by-line by the third party evaluators – and the level of rigour had

to be much higher than in the compiler case. Also, we were using the proof to understand the consequences of the specification and to debug the protocol, so it had to pass through a brain. We did the proof by hand, with most of the detail present (no use of 'similarly', and all branches done in full), with some intermediate steps accompanied by an intuitive explanation. We discovered, somewhat to our dismay, that we could not prove our refinement using the traditional Z refinement 'forward' or 'downward simulation' proof obligations given in [Spivey 1992, section 5.6]. So we had to go back to first principles, and derive the Z form of the other 'backward' or 'upward simulation' proof obligations. We derived these rules with a very high degree of rigour, since they provided the basis for our subsequent refinement proof.

3.3 What to prove

It is important to make sure all your valuable proof effort is being used to prove the right thing. For example, if your key security property is not preserved by refinement, then why are you doing a refinement proof? Or worse still, if your specification is inconsistent, you can 'prove' anything.

The following can be worthwhile things to prove:

- **Consistency checks**: Prove that your specification is consistent, that it has a model.
- **Sanity checks**: In a 'State and Operations' style specification, prove that the state constraint can be satisfied, and that the precondition of each operation is not *false*. More sophisticated sanity proofs include calculating the reachable states.
- **Emergent properties of a single specification**: Make explicit as a theorem some desired or suspected property or consequence of the specification, then prove it holds. Such theorems can be posed as 'challenges', in order to gain a deeper understanding of the specification. (Conjectures that turn out to be true are not nearly as interesting as ones you believe to be true, but that turn out to be false; with false ones you *learn* something – there is an error in your specification, or an error in your understanding of the *consequences* of the specification. Bold challenges are more likely to be false.)
- **Required properties of a single specification**: If some property is required to hold of the specification, and the specification has captured it only implicitly, it needs to be made explicit and shown to hold. (For example, it might be a requirement that some critical state can be reached only by passing through certain other states: if this is not captured in the state constraint, but is an emergent property of the operations, it will need to be proved.)
- **Properties across specifications**: Prove that a certain relationship holds between two specifications. The most common is the refinement relationship. Also, it might be necessary to prove that they both have the same key property, especially if this property is not preserved by refinement.

Our compiler proof is essentially a correctness proof: that any compiled program, expressed as a sequence of Asp instructions, has the same semantics as the original Pasp program. The specification is not in 'State and Operations' style, but rather is of various functions (*meaning functions* from syntax to semantics, and *compiler functions* from syntax to sequences of instructions). So the proof is that two functions are equivalent, that

$$\forall \text{ prog} \bullet M_{\text{Pasp}} \text{ prog} = (M_{\text{Asp}} \circ Comp) \text{ prog}$$

The purse specification is written in a more conventional 'State and Operations' style. Some of the security properties required of the abstract specification are implicit, and so needed to be proved to hold. Fortunately, all the required properties are preserved by refinement, so we then had to perform just a refinement proof between our abstract and concrete models.

3.4 How to prove it

You cannot just write two large specifications, and only then decide to do a refinement proof between them, except in the most trivial of cases. The fact that you are to prove something affects how you write the specifications. It is not a 'waterfall' process of specification, then proof, then implementation; it is very iterative, with things you learn in doing the proof feeding back into the specification. And there is a balancing act between the needs of proof, the needs of specification clarity, and the needs of implementation.

If the development itself is going to be incremental, with more functionality to be added later, it can be worthwhile spending time up-front getting a good structure for the proof that will ease later rework.

If you are using a particular tool, you may decide to write your specification with its limitations in mind. For example, if the tool is particularly lacking in support for some part of Z, such as free types or the more exotic schema calculus operators, you might decide to minimise the use of them. (Or you may decide to use a better tool.)

Our compiler development turned out to be incremental, but unfortunately we did not know that from the start. So we spent rather more time reworking the specification and proof structure than we might have. Also, the implementation step requires a clear translation between the Z specification and the Prolog implementation, which affected the way we wrote the Z: we used a rather more constructive style than a Z purist might like.

For the purse, in order to express some of the abstract security properties, we wanted a single schema that captured the entire operation of the device, which we got by disjoining all the individual operation schemas. In order for this disjunction to make sense, we had to make all the different operations' inputs the same type, and similarly for the outputs. We achieved this by using free types, which in turn made some of the proofs slightly trickier. The whole specification and proof task fell into two phases, with some functionality peripheral to the security being added in the second phase. We used this opportunity to structure our proof better, by extracting some of the commonality into lemmas (ensuring that these lemmas also made sense at an intuitive level). This allowed us to simplify the presentation of all the proofs, and to considerably simplify the two most complicated cases.

3.5 Structuring a large proof

When a proof runs to several hundred pages, it cannot be presented as an unstructured whole if it is to be understood (either by a third party, or by the original proof team a while later). It must be given a structure, broken down into manageable, comprehensible chunks. But once it has been broken down into parts, there is a danger that things will be missed, lemmas applied but not proved, branches forgotten.

So an overview that describes the structure, and makes it clear that everything that needs to be proved has been proved, is essential.

The compiler proof has an obvious structure that follows the specification: structural induction over the abstract syntax.

The largest part of the purse proof is the refinement proof, which naturally breaks up into proofs for each individual sub-operation. The various lemmas extracted in the second phase were proved separately, some in an Appendix, then invoked as appropriate. We also provided a two-page overview, which summarised as a structured tree all that needed to be proved, and cross referenced where each particular proof could be found in the document.

4. Experience with proof tools

Although all our proofs were originally done by hand, we have recently been experimenting with proof tools.

Stringer-Calvert has pushed the compiler case study from [Stepney 1993] through the PVS tool [Stringer-Calvert *et al*. 1997]. PVS does not understand Z as a native language, so a fair bit of conversion effort was needed, in particular the need to totalise all the partial functions. The conclusion was that the hand proof was essentially sound, but with some hand waving in the induction glue.

I have been pushing one small branch of the purse proof through CADiZ, a Z proof assistant [Toyn 1996] that performs the proof under close human guidance. CADiZ currently understands ISO Standard Z [Toyn 1998], not the 'Spivey Z' [Spivey 1992] that we used for our work, so a small amount of editing of our original LaTeX source was needed. No errors were found in the hand proof, but using the tool did suggest some clearer ways of presenting some details of the proof. Also, many of our correct but unstated assumptions about Z toolkit operators were exposed.

Another member of our team has been evaluating Z/Eves [Meisels & Saaltink 1997] by looking at another small branch of the purse proof. Again, no errors were found in the hand proof. In order to guide the proof, each of our intermediate hand proof steps was used as a lemma to prove the next: the tool managed to prove each of these steps quite straightforwardly. Interestingly, although the authors of Z/Eves say that they have yet to find a Z specification that passes all of their tool's precondition checks, we found that our specification did. We are not aware of having done anything special to ensure this, but it may be that the need to do proof made us much more careful in writing our specification in the first place.

One thing was apparent about *all* these investigations: a lot of time was spent 'fighting' the various tools, in different ways. Proof tools still have a long way to go before they provide the transparent support to the development process that, say, compilers or word processors give other users now.

5. My ideal Z proof assistant

Based on my experience of doing proofs by hand, and using a few tools, I have some requirements for my ideal proof assistant. Firstly, c*orrectness* is a given – I do not need a tool to help me make mistakes! Secondly, I do not want a fully automated tool

into which I feed conjectures and get the response 'true' or 'not true, because…'; I need these proofs to pass through my brain somehow.

So, I want an assistant that can help me do my proofs more quickly, less tediously, and more accurately than by hand, but still involve me deeply in the process.

5.1 User interface

My number one priority is the user interface. When I am doing a proof, I need to be immersed in the mathematics. I want to see Z symbols on the screen, not some mark-up language.

The tool should be *transparent*. I want to be aware only of the Z I am manipulating, not the way the tool works behind the scenes. For example, I never want, when looking at a predicate like $P \lor Q \lor R$, to be told that I can manipulate the $P \lor Q$, but not the $Q \lor R$, just because the tool internally holds this construct as a left-associative tree.

I want to be able to manipulate the Z *directly*, much the way I directly manipulate characters, words and sentences with a word processor, but here with the underlying tool also understanding the mathematical manipulations [Bertot 1997]. For example, in a predicate like $\exists S \bullet P \lor Q$, I want to be able to grab hold of the disjunction and 'pull' it outside the existential quantifier, and have the tool automatically distribute it: $(\exists S \bullet P) \lor (\exists S \bullet Q)$.

5.2 Z-based

The tool should be Z based. It should understand Z on the surface; there should be no need to translate the specification into another language. And it should understand Z underneath – it should talk to me in Z, about the Z I can see, not in another underlying language, or about some transformed variant of my Z. I do not care if it achieves this by being a special purpose Z tool, or a general purpose proof tool with the Z layer completely hiding the underlying generic tool.

The interesting parts of the purse refinement proof, in particular, involve manipulations of large nested schemas. The key parts of the proof involve extracting just the right parts from the schemas, and showing they have the right properties. By the time the expressions have been reduced to 'toolkit properties' [Spivey 1992, chapter 4], I want to be able to stop. I don't want to spend my life proving uninteresting low-level things, such as (these are all derived from real 'leaves' in my proof tool experiments)

$$(r \oplus \{ (x, y) \}) \, x = y$$
$$\langle \rangle \in \mathbf{F} (\mathbf{N} \times X)$$
$$a \in (X \leftrightarrow Y) \setminus \{ \varnothing \} \Rightarrow a \neq \varnothing$$
$$\# \langle a \rangle > 0$$
$$x = a \Rightarrow x \neq b \qquad \text{where } F ::= a \mid b \mid \ldots$$

So the tool should know more than just core Z – in particular it should have a very rich set of laws to reason about Toolkit operators, and about free types. It should be able to do all the above, and more, automatically, telling me only about any subtle side conditions that I might have forgotten.

5.3 Navigation, structure, and presentation

One of the problems with a large proof is getting lost in the detail, being unable to find other parts, being unable to see the structure. Some kind of navigation aid is needed – both for constructing the proof, and for understanding its structure and evaluating it later. Some kind of 'zoomed out' view of a proof is needed.

The tool should also provide intelligent support for building the structure in the first place, in particular, help in forming useful lemmas.

There should also be some support for presenting a printed overview of the proof, with the key stages, and the key manipulations that were used to get between those stages.

5.4 Replay and 'Similarly'

Reworking is one of the most frustrating things about hand proof. The proof is passed through a brain and understood, then some small change is made to the specification, and the proof has to be reworked. So my ideal assistant would be able to abstract the structure of the proof from my detailed mouse clicks, and replay it to do the 'same' proof on a different specification.

And similarly, if I have a proof with several branches, each only slightly different from the others (the same key manipulations parametrised by the Z operation, say), I would like to be able to say 'look at that branch – now do the analogous thing to this branch'.

Of course, I realise that deciding what the similarity or analogy is, is an interesting, highly non-trivial problem [Hofstadter *et al*. 1995].

6. Summary

Specification keeps you honest, proof more so, and proof tools even more so. The more effort applied to proof, the higher the assurance that can be achieved. But the current generation of proof tools is not yet up to performing industrial-scale proofs. The user interface is the most important area in need of improvement.

7. Acknowledgements

I would like to thank Dave Thomas and Wilson Ifill of AWE, and the NatWest Development Team, for providing such interesting Z projects to work on. Also, my thanks to University of York, and Ian Toyn in particular, for providing me with the facilities to experiment with CADiZ.

8. References

[Bertot 1997]
Y. Bertot. Direct manipulation of Algebraic Formulae in Interactive Proof Systems. UITP'97. http://www.inria.fr/croap/events/uitp97-papers.html

[DeMillo et al. 1979]
R. A. De Millo, R. J. Lipton, and A. J. Perlis, Social Processes and Proofs of Theorems and Programs. CACM, 22(5), 271--280. 1979.

[Hofstadter *et al*. 1995]
Douglas R. Hofstadter and the Fluid Analogies Research Group. *Fluid Concepts and Creative Analogies: computer models of the fundamental mechanisms of thought*. Penguin. 1995.

[Meisels & Saaltink 1997]
Irwin Meisels and Mark Saaltink. *The Z/EVES Reference Manual*. TR-97-5493-03c. June1997. http://www.ora.on.ca/z-eves/

[Spivey 1992]
J. M. Spivey. *The Z Notation: a reference manual*. 2nd edition. Prentice Hall International Series in Computer Science. Prentice Hall. 1992.

[Stepney 1993]
Susan Stepney. *High Integrity Compilation: a case study*. Prentice-Hall. 1993.

[Stepney 1998]
Susan Stepney. Incremental development of a High Integrity Compiler: Experience from an industrial development. In *Third IEEE High-Assurance Systems Engineering Symposium (HASE'98),* Washington DC 1998.

[Stepney *et al*. 1998]
Susan Stepney, David Cooper, and Jim Woodcock. More powerful Z data refinement: Pushing the State of the Art in industrial refinement. In J. P. Bowen et al., editor, *ZUM'98: 11th International Conference of Z Users*, Berlin 1998. Lecture Notes in Computer Science. Springer-Verlag. 1998.

[Stringer-Calvert *et al*. 1997]
David W. J. Stringer-Calvert, Susan Stepney, and Ian Wand. Using PVS to prove a Z refinement: a case study. In *FME '97: Formal Methods: Their Industrial Application and Strengthened Foundations*, Graz, 1997. Lecture Notes in Computer Science. Springer-Verlag. 1997.

[Toyn 1996]
Ian Toyn. Formal reasoning in the Z notation using CADiZ. In N. A. Merriam, editor, *2nd International Workshop on User Interface Design for Theorem Proving Systems*. Department of Computer Science, University of York, July 1996.
http://www.cs.york.ac.uk/~ian/cadiz/home.html

[Toyn 1998]
Ian Toyn (project editor). ISO Standard Z Notation. 1998. http://www.cs.york.ac.uk/~ian/zstan/