More Powerful Z Data Refinement: pushing the state of the art in industrial refinement

Susan Stepney[1], David Cooper[1], and Jim Woodcock[2]

[1] Logica UK Ltd, Betjeman House, 104 Hills Road, Cambridge, CB2 1LQ, UK
stepneys@logica.com cooperd@logica.com
[2] Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK
Jim.Woodcock@comlab.ox.ac.uk

Abstract. We have recently completed the specification and full refinement proof of a large, industrial scale application. The application was security critical, and the modelling and proof was done to increase the client's assurance that the implemented system had no design flaws with security implications. Here we describe the application, and then discuss an essential lesson to learn concerning large proof contracts: that one must forge a path between mathematical formality on the one hand and practical achievement of results on the other. We present a number of examples of such decision points, explaining the considerations that must be made in each case.

In the course of our refinement work, we discovered that the traditional Z data refinement proof obligations [Spivey 1992b, section 5.6], were not sufficient to prove our refinement. In particular, these obligations assume the use of a 'forward' (or 'downward') simulation. Here we present a more widely applicable set of Z data refinement proof obligations that we developed for and used on our project. These obligations allow both 'forward' and 'backward' simulations, and also allow non-trivial initialisation, finalisation, and input/output refinement.

1 Introduction

Over the past few years we have been working with the NatWest Development Team proving the correctness of Smartcard applications for electronic commerce. Here we describe one of those applications. We have modelled the abstract behaviour of the product, modelled its more concrete top level design, and have rigorously proved the correctness of the refinement from one to the other. All work was done in Z.

In the first half of this paper, we describe the application and the form of the design step modelled (section 2), and discuss a number of lessons learnt during the proof process (section 3). These lessons centre around the tension between mathematical formality and the need to achieve a demonstrable benefit to the project.

In the second half of this paper (sections 4–7), we focus on one particular lesson: the importance of a sufficiently rich refinement theory. We discovered that the traditional Z data refinement proof obligations [Spivey 1992b, section 5.6] (hereafter we refer to these as 'the Spivey rules') were not sufficient to prove our refinement, because they make some simplifications, and assume the use of a 'forward' (or 'downward') simulation. We had to remove some of these simplifications, and we required the use of a 'backward' simulation. We present a more widely applicable set of Z data refinement proof obligations that we developed for and used on our project. These obligations allow both 'forward' and 'backward' simulations, and also allow non-trivial initialisation, finalisation, and input/output refinement.

2 The application

NatWest Development Team had a product under development that was deeply security critical. They were developing a Smartcard application to handle electronic commerce, and they wanted to be sure that these cards would not contain any bugs in implementation or design that would allow them to be subverted once in the field. They called in Logica to develop formal models of the system and the security policy, and to prove that the system design met all the security properties required.

The system consists of a number of *electronic purses* that carry financial value, each hosted on a Smartcard. The 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.

These cards are to be sold to members of the public to enable them to carry out fully electronic financial transactions with other individuals, with banks, retailers, etc. Insecurities could allow people to forge the value on their purses, and so obtain goods for free, thereby severely impacting the commercial viability of the project.

The task of the formal methods team was to model the system to ensure it behaved sensibly, and prove that the system design accurately reflected the behaviour specified.

2.1 Models

We developed two key models. We wrote an *abstract* model, describing the world of purses and the exchange of value through atomic transactions. This model expressed the security properties that the cards must preserve.

We wrote a *concrete* model, mirroring the design of the purses, which exchange value using a protocol of messages.

We rigorously proved that the concrete model is a *refinement* of the abstract.

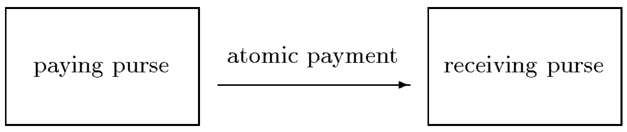

Fig. 1. An atomic transaction in the abstract model.

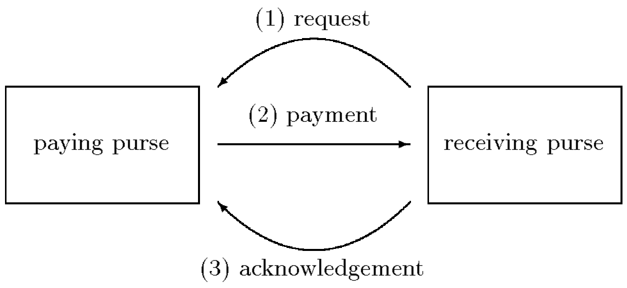

Fig. 2. Part of the n-step protocol used to implement the atomic transaction in the concrete model.

Abstract model: The abstract model is small, simple, and easy to understand, running to approximately 20 pages of Z and natural language commentary.

The key operation is to transfer a chosen amount of value from one purse to another, modelled as an atomic action to decrement the value in the paying purse and increment the value in the receiving purse (figure 1). This operation (and all the others, too) preserves the two key system security properties:

– no value may be created in the system
– all value is accounted in the system (no value is lost)

The simplicity of the abstract model allows these properties to be expressed in a way that the client can easily see are as they desired them to be. 20 pages of Z may sound a lot for such a simple operation, but the abstract model also includes a number of other operations around the periphery of this central behaviour, such as card locking and query operations.

Concrete model: The concrete model is more complicated, reflecting the real system design. The key changes from the abstract are:

– transactions are no longer atomic, but instead follow an n-step protocol (figure 2)
– the communication medium is insecure and unreliable
– transaction logging is added to handle lost messages
– no global properties: each purse has to be implementable in isolation

This model is approximately 60 pages long, including Z and natural language commentary. Once again, there are a number of periphery operations and state components in addition to those to model the central operation of value transfer.

The basic protocol is:

1. communication device ascertains the transaction to perform
2. receiving purse requests the transfer of an amount from the paying purse
3. paying purse sends that amount to the receiving purse
4. receiving purse sends an acknowledgement of receipt to the paying purse

The protocol, although simple in principle, is complicated by several facts. The protocol can be stopped at any point by removal of power from a card; the communication medium could lose a message; a wire tapper could record a message and play it back to the same or different card later. In the face of all these possible actions, the protocol must correctly implement the atomic transfer of value as specified in the abstract model.

2.2 Proofs

All the security properties of the abstract model are *functional*, and so are preserved by refinement. It is well known in security circles that some properties (such as information flow properties) are not, in general, preserved by refinement. We were saved from such complications in this case, and could concentrate just on proving refinement between the models.

The purpose of performing the proof is to greatly increase the assurance that the chosen design (the protocol) does, indeed, behave just like the abstract, atomic transfers. We chose to do rigorous proofs by hand, because our experience of existing proof tools is that current tools are not yet appropriate for a task of this size. We did, however, type-check the statements of the proof obligations and many of the proof steps using a combination of fuzz [Spivey 1992a] and Formaliser [Flynn *et al.* 1990] [Stepney]. As part of the development process, all proofs were also independently checked by external evaluators.

The proofs of the refinement obligations and the proofs of some model consistency obligations take approximately 200 pages. In addition, we produced approximately 100 pages of formal derivation in support of the underlying theory.

3 Between the devil and the deep blue sea

In developing the models and the proofs we were caught between opposing forces. On the one hand, we had a real product that was to hit the streets. We were proving properties because the client wanted to be sure the design was secure, not because they were a fan of obscure mathematical theories. We had to get the job done. On the other hand, proofs would give us no added assurance if they were faked, rushed, or so inelegant that no one would be able to check them.

The formal aspects of the development therefore plotted a course between perfect mathematical formality on one side and pragmatic 'just do it' on the other. Our course weaved from side to side as different concerns appeared. At each stage we focused on attacking the weakest part of the formal argument. Sometimes it was lack of faith in the questions we were asking, which forced us toward greater mathematical formality in search of an underlying theory to support our decisions. At other times it was the need to press on and cover more of the purse functionality, forcing us to 'just do' the proofs any way we could.

It is important to see this process as a continual trade off. There is no point in having a very sound underlying mathematical theory, but then no time or money to apply it to the product. Equally, there is no point in proving a theorem about the product when you have no understanding of whether it is an appropriate theorem to be proving.

There were a number of specific examples of this trade off, discussed below.

3.1 New proof rules needed

One of the early problems with the development forced us in the direction of formality to solve it.

We had developed an abstract model, with a single atomic transaction, and a concrete model with the protocol steps. Intuitively it appeared that the concrete should be a refinement of the abstract, but all our attempts to prove refinement using the Spivey rules failed. Looking at why the proof failed showed that there was a serious obstacle, not just an inability to push symbols around. The problem centred on when non-determinism was resolved: the concrete protocol resolved the non-determinism inherent in the system later than did the abstract atomic transaction. The rules of refinement that we were trying to use allow non-determinism to be resolved *earlier* in the concrete, but not later. (See section 4.2 for more detail.)

We made some brief attempts to 'just do it' anyway, trying to modify the models to resolve the non-determinism differently, and even trying to prove something other than refinement. But it became clear that a more fundamental look at the proof rules held better hope for a solution.

There is a larger, more general theory of refinement, of which the Spivey rules are a specific instance, the forward proof rules. The more general theory includes another set of rules, the backward proof rules, which cater for the form of non-determinism resolution we needed. We needed to recast these rules from their relational form into the Z world of schemas, state transitions, inputs and outputs. (We have subsequently published a simplified form of these newly derived Z backwards rules in [Woodcock & Davies 1996]. See section 6 for more details of the derivation of the full proof rules we derived, including input-output refinement.)

Armed with an early version of the new rules, we were able to carry out our proofs successfully. However, as we did so, we found the new rules seemed too good, making our proofs too easy. We started investigating toy examples, and found we could prove refinement of patently non-refining systems! Rushing back in the direction of mathematical formality, we discovered that yes, indeed, these early proof rules were unsound, and some other aspects that we had originally thought unnecessary (because they were unnecessary in the forward rules) needed to be brought in. Some more working to put all the proof rules on a sound footing, doing all the derivations in detail, gave us the confidence to go back to our models, knowing that we were now working in the right context.

So, the path we took in this case was

1. 'just did it' with the theory we had until the theory (the proof rules available to us at the time) failed us
2. moved toward formality to find a larger, more general theory that included some new tools (backward rules) to help
3. used the new rules until doubt arose
4. investigated the problems (toy examples), then went back to correct
5. tightening of the theory
6. use the new theory with confidence

3.2 Problems with generic proofs

Sometimes doing things elegantly costs too much.

We had first modelled and proved a reduced version of the system, and were now expanding it to incorporate the full richness of the actual system. As we added one particular feature, we noticed that it had a similar mathematical structure to a feature already modelled and proved. The existing feature used integer addition as a binary operator, and the new feature used *max*, but the proofs seemed to rely only on properties of addition that are shared by *max* (such as commutativity).

Rather than just blindly re-doing all the proofs with the new feature, we decided to generalise the existing proof, and then separately instantiate it, once for addition and once for *max*. This should have cut down on the amount of work (one set of proofs rather than two) and made it easier for the evaluators to understand (understand one general proof, rather than two 'similar' ones).

As most people who have worked on automated theorem provers probably know, generalising proofs is much harder than it looks. For example, addition has an inverse, and we had used subtraction to simplify our original proof, even though it was not necessary to do so. To generalise the proof so that it was also applicable to an operator like *max* without an inverse was possible, but made the proof more complicated.

In other places we made use of properties of addition that are shared by *max*, but not by all binary operators. Thus we had to decide what the key property was so that the scope of the generality of the proof could be defined.

All this meant our general proof was becoming significantly longer than the original, and very obscure. There seemed to be no intuitive peg on which to hang our understanding — we could read the proof only as a series of meaningless symbols.

Although we saw no theoretical reason why a general proof would not be possible, we abandoned the attempt. We needed a proof of both features: we had one already for addition, and our work on the general proof had shown how the proof of the *max* feature would go. We therefore pressed ahead on a specific proof for the second feature, finishing it in less time than it would have taken us to finish the general proof and two instantiations. Furthermore, we found the proofs actually easier to understand as specifics than as instantiations of the general proof.

We believe it would have been possible to complete the general proof. It may well be possible to complete it more elegantly, and produce instantiations that are intuitively understandable. But in this case we could gain the benefit from the proof (increased assurance) for less effort by doing two, similar, repeated proofs.

Our weaving path was

1. do a specific proof
2. notice the scope for generality and elegance, and start a general proof
3. abandon general proof when cost grows too large
4. use the experience gained doing the general proof to 'just do' two specific proofs

3.3 No need to justify a working strategy

Even if you think there may be a better solution around the corner, if you have a solution that works, that may be good enough. This is a lesson hardest to accept if you are an academic at heart. The nature of academic research encourages reworking and revisiting solutions to find different insights. But in an industrial context, the benefit of such further work may not be worth the cost.

We had been using our backward proof rules to prove the refinement of the abstract model by the concrete model. We were successful with some proofs, but there were always some proofs at which we failed. We modified the models, often making the failed proofs possible, but then invalidating the previously successful ones.

The problem centred on the use of the backward proof rules. In the more conventional forward Spivey proof rules you attempt to show that there is a reasonable abstract state that an operation can take you *to*, given that you have an abstract state to start from and a concrete operation that occurs. In the backward rules, you instead attempt to show that there is a reasonable abstract state that an operation can take you *from*, given that you have an abstract state to go to and a concrete operation that occurs. This requires you to reason about the properties of a previous state.

Our model makes such reasoning difficult, because as the protocol progresses information that is no longer needed is lost from the state, and it is precisely this information that is needed in order to construct the prior abstract state. We could not include this extra information in the concrete model because it

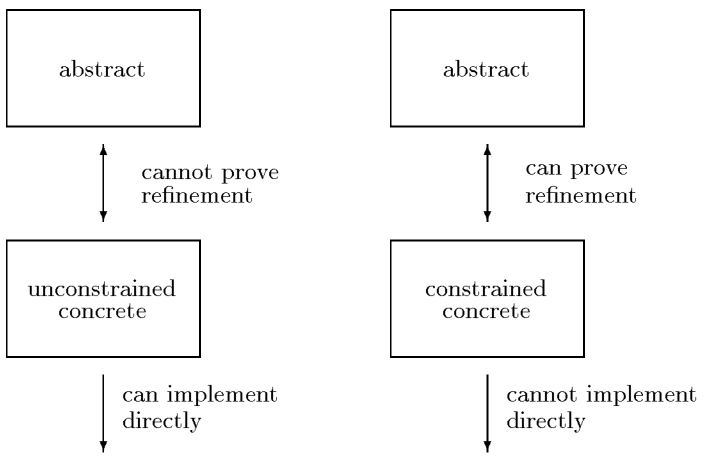

Fig. 3. We found that we could implement, but not prove, or that we could prove, but not implement.

relates to global properties of a number of purses, which cannot be implemented by a set of independent purses.

However, we knew that these global properties *do* hold: the protocol forces them to hold. It is just that a model that expresses them explicitly cannot be implemented appropriately. If these constraints are in the concrete model, the refinement can be proved, but the model does not match the implementation. If these constraints are *not* in the concrete model, the model matches the implementation, but we cannot prove the refinement (figure 3).

The solution was to add the constraints in an intermediate model, and then do an additional refinement to remove them (figure 4) from the final model.

This second refinement could be performed using the conventional forward rules. This we did, and succeeded in proving both refinements. This issue drove the entire structure of the development.

This approach raised a question: does there exist a single refinement that could do the job of these two? Although this question is interesting, as far as our project in hand was concerned *it was irrelevant*. We had a solution that worked (an intermediate model, with two refinements) that was mathematically sound. Whether or not two refinements are *necessary*, in this case two refinements are certainly *sufficient*. We ignored the forces pulling us toward mathematical elegance, and pressed on using a solution that worked, even though we did not have a theoretical justification for its necessity.

3.4 Rework repaid in easier understanding

In contrast to the previous two examples, we also found that sometimes reworking a successful solution more elegantly and generally can be worthwhile.

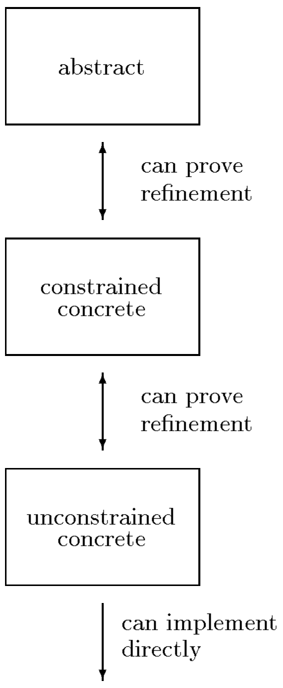

Fig. 4. We proved the refinement in two steps.

We developed our models and proofs in two phases: once concentrating on just the key central functionality, and then again covering all the functionality. Having done some parts of the derivations repeatedly for different but similar operations, when we came to the rework we extracted the similar parts, generalised, and defined a number of useful lemmas. These we used to simplify some of the derivations for the new operations.

While generalising the derivations, we noticed that those operations that aborted the current transaction did not fit into the mould of the generalised derivation. However, if the aborting part were extracted, the remaining piece of the operation did fit. This restructuring made the general derivation more widely applicable, and significantly simplified the proofs of the aborting operations.

So here our path was

1. develop complete models and proof for a subset of the system (just do it)
2. identify commonalities (mathematical elegance)
3. generalise
4. expand with additional functionality

3.5 Hand-proof delivered the goods

Given a client who wants to "increase assurance through formal methods", one has to use the tools to hand and supply the greatest possible increase in assurance at acceptable cost. It is not necessary to have every mathematical i dotted and t crossed before you can help real projects.

We decided to do all our proofs by hand, with no machine checking of the proof steps (although we did type-check all the mathematical statements). There are a number of good Z theorem provers/checkers around, such as Z/Eves [Meisels & Saaltink 1997], ProofPower [King & Arthan 1996], and CADi$\mathbb{Z}$ [Toyn 1996], but we believe the extra cost of doing proof at the level of rigour enforced by a theorem prover/checker far outweighs the benefits of greater assurance in this case. We have done some small scale investigations of the cost of using CADi$\mathbb{Z}$ and Z/Eves for our proofs that support this view.

However, it is likely that if a set of proofs need to be maintained in the face of continuing changes to a model, tool support may become more important.

3.6 Presentation is important

Having a mathematically correct proof is insufficient if it is presented too badly to be read and understood by a reviewer.

Part of the development process imposed by the client includes detailed external evaluation of the formal models and proofs. These evaluators have to be able to read a proof and both understand it intuitively and check it line by line for correctness. To ensure readability we developed a number of presentation styles.

We started all proofs with a clear, mathematical statement of the theorem to be proved, expressed in the conventional $\mathit{hypothesis} \vdash \mathit{conclusion}$ style, where *hypothesis* is a declaration and *conclusion* is a predicate. This rigour ensured that we knew clearly what was to be proved, and often prompted us to ask why we were proving this. We also described the theorem in English, explaining what property was being proved, and explaining why it was reasonable to believe that it was true.

Many proofs were broken down into a number of subsections, and in these cases we stated the subtheorems themselves formally and, where possible, intuitively justified them. To make it easy to follow this nesting of proofs, when a subtheorem was proved, we added an end marker $\Box$ labelled with the section number, to help bracket the subproof. For example,

```
3.1 theorem ...
    proof steps, leading to two things to prove ...
    3.1.1 first sub-theorem
          proof of first sub-theorem
          □ 3.1.1
    3.1.2 second sub-theorem
          proof of second sub-theorem
```

□ 3.1.2
tidy up of whole proof ...
□ 3.1

Lemmas that were of a more general applicability were extracted into an appendix, so they could be read in isolation.

We wrote the proofs themselves in one of two styles: either rigorously step by step, with each step labelled with an inference rule; or more free-flowing English with the key points expressed mathematically.

We used the rigorous presentation when symbol manipulation was most important, which often occurred when large schemas were being restructured and manipulated to extract some key component. This was the predominant style, and we used it whenever we were in doubt of the validity of our arguments.

The more free-flowing presentation was appropriate when there was little doubt of the validity of the argument, but the details would be long and cumbersome.

A formalist would demand all arguments to be presented step-by-step, with each step labelled. But with limited resources, it is sometimes better to tighten up the rigour of some other part (such as the derivation of the proof rules themselves) than spend the precious resource expanding "some set-theoretic manipulations can show ..." in detail.

4 A closer look at the need for new proof rules

As explained earlier (section 3.1), in the course of our refinement work it became apparent to us that the traditional Z data refinement proof obligations [Spivey 1992b, section 5.6] were insufficient for our purpose. We were forced back to first principles, to derive suitable new proof obligations.

Here we present the two sets of Z data refinement proof obligations we derived for our work, each of which are more more widely applicable than the Spivey rules. These new rules cover both 'forward' and 'backward' simulations, allow non-trivial initialisation and finalisation steps, and provide the ability to perform input/output refinement.

In the following sections we describe which assumptions used to derive the traditional rules we relaxed, and the new proof obligations we derived[1].

4.1 Traditional Z data refinement proof obligations – recap

The traditional Z data refinement proof obligations are given in [Spivey 1992b, section 5.6]. To recap, they comprise the three following proof obligations:

initialisation: for each concrete initial state, there must be a corresponding abstract initial state.

$CInit \vdash \exists A' \bullet AInit \wedge R'$

[1] We use Standard Z [Z Standard 1995] syntax in our small illustrative examples, and note where this differs from the Z Reference Manual [Spivey 1992b] syntax.

applicability: whenever it is possible to perform the abstract operation AOp, it must be possible to perform the concrete operation COp on the corresponding concrete state (also known as 'widening the precondition').

$$R; \text{pre } AOp \vdash \text{pre } COp$$

correctness: for any abstract state A within the precondition of AOp, corresponding to a before state C (which, by applicability, is within the precondition of COp), then, corresponding to any C' reachable from C by COp, there must be an abstract state A' related to A by AOp.

$$R; COp \mid \text{pre } AOp \vdash \exists A' \bullet AOp \land R'$$

4.2 Resolution of non-determinism

The Spivey rules are applicable when the concrete model resolves any remaining non-determinism sooner (or at the same point as) the abstract model. However, it is possible to develop models where concrete non-determinism is resolved later than the abstract non-determinism.

For example, consider the case of two booking offices [Woodcock & Davies 1996, section 17.3]. The Apollo theatre, when you book a ticket, chooses a particular ticket to give you, and when you arrive at the theatre, gives you that ticket. It non-deterministically chooses which particular ticket you get *early* in the process. The Phoenix cinema, on the other hand, when you book a ticket, merely notes that yet another ticket has been booked; only when you arrive at the cinema does it choose which particular ticket to give you. It non-deterministically chooses which particular ticket you get *late* in the process. These two systems are behaviourally equivalent — so each is a refinement of the other — but with the Spivey rules we can prove only that (early) Apollo refines (late) Phoenix, not that Phoenix refines Apollo.

Our own application was another, much larger, example of later resolution. We had an abstract transaction that atomically either succeeded or failed, straight away. The concrete model implemented the transaction as an n-step protocol, and whether the transaction succeeded or failed could not be determined until late in the protocol.

In all such later-resolution cases, the Spivey rules *cannot* prove the refinement. To see why this is, consider figure 5. This shows a non-deterministic abstract operation AOp_0, followed by a deterministic one AOp_1. In our application, AOp_0 is the initial atomic transaction that non-deterministically succeeds or aborts, and AOp_1 is essentially 'nothing happening', ΞA. AOp_0 is refined by a deterministic concrete operation COp_0, and AOp_1 is refined by a non-deterministic concrete operation COp_1. In our application, COp_0 is the first always successful *request* step of the protocol, and COp_1 is the later *payment* step that either succeeds or aborts.

By the time both operations have been performed, the non-determinism has been resolved both concretely and abstractly, and so the retrieve relation relates

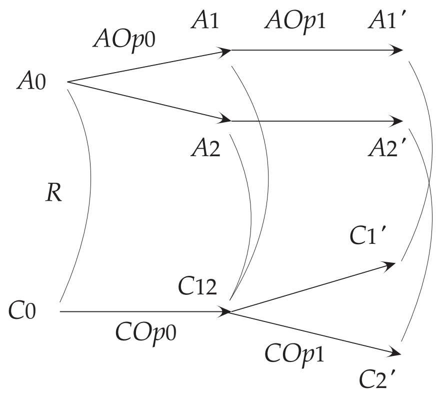

Fig. 5. A case not provable by the Spivey rules: non-determinism resolved later in the concrete than in the abstract.

A_1' and C_1', and separately relates A_2' and C_2'. Because the first operation can result in one of two abstract states, A_1 and A_2, but only a single concrete state, C_{12}, the retrieve relation has to relate C_{12} to both A_1 and A_2.

Now consider what happens when the Spivey rule for correctness is applied to the second operation, AOp_1 and COp_1:

> *for any abstract state A within the precondition of AOp* (here consider the state A_1), *corresponding to a before state C* (which is C_{12}) *then, corresponding to any C' reachable from C by COp* (let's consider C_2'), *there must be an abstract state A' related to A by AOp.*

But there is *no* such state A' that retrieves from state C_2' whilst being reachable from state A_1 by the abstract operation. The proof fails.

It turns out that the Spivey rules are *sufficient* — anything they can prove is indeed a refinement — but not *necessary* — there are some cases of refinement that cannot be proved using them. We discovered that our own application fell into this second class, and so we needed to understand refinement better in order to derive refinement proof obligations that were sufficient for our case.

5 What is refinement?

There are some specifications that we intuitively feel *are* refinements, but which we cannot prove using the Spivey rules as they stand. So, what *is* a refinement?

5.1 A relational model

[He *et al.* 1986] give a *relational* definition of refinement. A 'global-to-global' relation can be *implemented* by moving into some abstract world (by a process called initialisation), performing a sequence of abstract operations, then moving

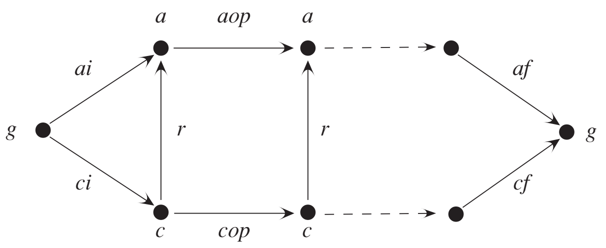

Fig. 6. A relational view of refinement (after [He *et al.* 1986]). The global-to-global relation $g \longleftrightarrow g$ can be implemented abstractly, by initialisation ai, operation(s) aop and finalisation af. The corresponding concrete relation, defined by ci, cop, and cf, refines the abstract precisely when it is a subset of the abstract relation.

back into the global world (called finalisation). If another implementation (via a concrete world) defines a relation that is a subset of the one obtained via the abstract world, then the concrete implementation *refines* the abstract one. (We refer the reader to the original paper for precise definitions of these terms, sketched in figure 6.)

This subsetting requirement formally captures the notion of 'refining away non-determinism'. The [He *et al.* 1986] definitions require all the various global, abstract and concrete operations to be *total*, so there is no danger of subsetting all the way to the empty set.

We can now ask the question, 'what is a refinement?' independent of the particular choice of refinement proof obligations. Refinement, in this relational world, is simply subsetting.

Reasoning over the whole global relation, which itself is defined in terms of sequences of operations ('programs' in the terminology of [He *et al.* 1986]) is difficult. It is much more tractable to reason over individual operations, and [He *et al.* 1986] give *two* different sufficient conditions for a collection of concrete operations to be a refinement of a collection of abstract operations in this relational world. The Spivey rules are derived from one of these conditions, that for 'forward simulation'. However, the other condition, for 'backward simulation', is the one appropriate for proving refinements with a later resolution of non-determinism.

5.2 Casting to the Z world

Now that we know what a refinement is in the relational model (expressed in terms of total operations, with no concept of inputs or outputs), we need to convert to the Z world (of partial operations, inputs, and outputs), and thereby translate the relational refinement obligations into corresponding Z ones.

Making certain simplifying assumptions (see section 6), and translating one of the [He *et al.* 1986] conditions, that of 'forward' simulation, results in the Spivey rules. If we make the same simplifying assumptions, and translate the other

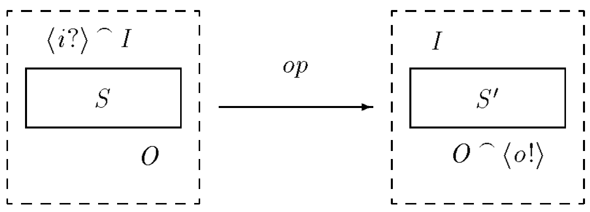

Fig. 7. Embedding a Z-like state S, input sequence I and output sequence O in the relational state, and imposing a computational model on the behaviour of I and O.

'backward' simulation condition, we get the analogous 'backward' Z refinement rules. (We have since published these simplified backward rules in [Woodcock & Davies 1996, section 17.3].) If, further, we relax some of the simplifying assumptions, we get the more powerful refinement rules (see section 6) that we found necessary for our particular application.

The new (simplified) backward rules allow us to prove that the situation in figure 5 is indeed a refinement. Formally, we have to show:

$$COp ; R' \vdash \exists A \bullet R \wedge AOp$$

Informally:

> *for any abstract state* A' (consider state A'_2) *corresponding to the after state* C' (which is state C'_2) *reached from* C (which here is state C_{12}) *by the concrete operation, there must be an abstract state* A (there is! — it is state A_2) *that both corresponds to* C *and is related to* A' *by the abstract operation.*

6 Relaxing the simplifying assumptions

Now that we have the wider theory from [He *et al.* 1986] available to us, we can see some of the *other* assumptions (beyond the choice of forward simulation) that go into deriving the Spivey rules, and consider relaxing those assumptions, too. The proof obligations that result from relaxing some of these assumptions are given in section 7.

6.1 Computational model

The wider theoretical model of [He *et al.* 1986] is purely relational; it has no concept of inputs or outputs. When moving to a Z model, the relational state needs to be provided with some internal structure, to give it a Z-like state along with (sequences of) inputs and outputs.

A *computational model* is then imposed, determining restrictions on how these input and output sequences can be used in operations. We choose a natural one (others are possible): each operation consumes the head of the input sequence, and appends an output to the output sequence (see figure 7). Furthermore, we require the operation itself to depend only on this input and output (and the state): it is independent of the 'past outputs' and 'future inputs' sequences.

6.2 Observability and Finalisation

It is well known that the refinements of a specification provable using the Spivey rules depend on what parts of the state are 'observed' by outputs. For example, if a *Stack* specification includes no operation such as *Top* to observe the state, then bizarre specifications — such as ones that randomly change the values, or throw them away altogether — can be proved to be refinements. In the trivial case where all outputs are indistinguishable then any specification can refine any other (with identically named operations), by chosing the chaotic retrieve[2] $R == [A; C \mid true]$.

In [He *et al.* 1986]'s relational model, the *finalisation* process captures what is observable. When deriving the Spivey rules, the only part of the relational state that is finalised is the output sequence. This is why, with those rules, the outputs provide the only way of observing the state.

In an abstract model, state is usually present because it is felt to be needed to capture the required abstract properties. It is certainly possible to have a 'large' abstract state that is not observed, in order to ease the statement of these properties. (For example, consider a 'Parity' specification that keeps the entire sequence of input bits in the abstract state, but where only the parity of this sequence is observed.) Providing 'observation' operations that merely observe the state may well pollute the model, especially if these operations do not map naturally onto any concrete operation. In such cases, a cleaner model may be possible by finalising (part of) the state to the relevant global state, and thus observing it directly.

In our own application, our abstract state includes a component, *sink*, that records the quantity of the value that has become unavailable to the rest of the system. A property of *sink* is quoted as a security requirement: the total value still available to the system, plus that in the *sink*, is required to be constant. The concrete model, following the actual implementation, has the corresponding information distributed amongst the collection of automonous purses, in various log files. The retrieve relation relates all these log files in a subtle way to the abstract *sink*. We showed that the distributed log files implement the abstract *sink*, and thereby satisfy the required security property. There is a concrete operation that observes the contents of a single purse's log file, but we did not wish to clutter our model with a concrete operation that corresponds to observing all the log files simultaneously, which would correspond to observing the abstract *sink*. So, instead of observing this component with an output, we finalise the *sink* component of the abstract state, and the *log* components of the concrete state, and discharge a finalisation proof obligation.

[2] In Standard Z [Z Standard 1995] notation, schemas are introduced using '==', rather than '$\widehat{=}$'.

6.3 Input/output refinement

With the Spivey rules, the concrete and abstract (and global) models use the *same* input and output state; only the Z-state part of the relational state is allowed to differ. Initialisation of the inputs (how the inputs in the abstract or concrete model are related to the global ones) is a trivial identity, and the input initialisation proof obligation vanishes, leaving only the state initialisation rule. Similarly, finalisation of the outputs is a trivial identity, and finalisation of the state throws it all away, and so the entire finalisation proof obligation disappears.

As explained above, we relaxed the assumption about finalising the state to nothing, to allow non-trivial state finalisation. We also relaxed the assumption that the inputs and outputs are the same in all models, allowing the concrete and abstract inputs and outputs to differ (from each other, and from the global ones, too). So now we can refine our input and output data types from more abstract to more concrete representations, in the same way that we have always been able to refine state. This requires an additional definition of how inputs are initialised and how outputs are finalised.

In the relational world, the retrieve relates the entire abstract and concrete state (including inputs and outputs as well as the Z-like state). With the Spivey rules, where the input and output parts are the same in the concrete and abstract models, there is no need to retrieve them (the retrieve is just the identity). Relaxing this assumption of equality, we now have to provide a part of the retrieve relation to map concrete inputs and outputs to abstract ones.

We choose to require that the complete retrieve be written as three independent parts: one between inputs, one between outputs, and one between Z-states. This does restrict the kinds of refinements that are provable. For example, anything observed by finalisation abstractly has to be observed by finalisation concretely; the requirement of independence of the retrieves means that we cannot move the observation into an output. However, we found that the resulting proof obligations, even with this restriction, are sufficiently powerful to enable us to prove our own application, and so any more general rules would have been overly complicated.

As an example of output refinement, consider a global model of traffic lights where the output is the current colour state, represented as an element of an enumerated (free) type; the abstract model could use natural numbers to represent these colours, and the concrete model could say how these numbers are represented as a sequence of bits.

$$LIGHT ::= red \mid amber \mid green$$

$$aLight == \{0, 1, 2\}$$

$$cLight == \{\langle 0, 0\rangle, \langle 0, 1\rangle, \langle 1, 0\rangle\}$$

The global model talks in terms of global outputs $g! : LIGHT$, the abstract model in terms of $a! : aLight$, and the concrete model in terms of $c! : cLight$.

The abstract output finalisation defines how the abstract numbers represent the global colours:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad AFin_! \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
a! : aLight \\
g! : LIGHT \\
\hline
g! = red \Rightarrow a! = 0 \\
g! = amber \Rightarrow a! = 1 \\
g! = green \Rightarrow a! = 2 \\
\hline
\end{array}
$$

We can write this schema equivalently as[3]:

$$
\begin{aligned}
AFin_! == \{ & \langle\!| g! == red, a! == 0 |\!\rangle, \langle\!| g! == amber, a! == 1 |\!\rangle, \\
& \langle\!| g! == green, a! == 2 |\!\rangle \}
\end{aligned}
$$

The concrete output finalisation defines how the concrete bitstreams represent the global colours:

$$
\begin{aligned}
CFin_! == \{ & \langle\!| g! == red, c! == \langle 0,0 \rangle |\!\rangle, \langle\!| g! == amber, c! == \langle 0,1 \rangle |\!\rangle, \\
& \langle\!| g! == green, c! == \langle 1,0 \rangle |\!\rangle \}
\end{aligned}
$$

The output retrieve defines how the concrete bitstreams represent the abstract numbers:

$$
\begin{aligned}
R_! == \{ & \langle\!| a! == 0, c! == \langle 0,0 \rangle |\!\rangle, \langle\!| a! == 1, c! == \langle 0,1 \rangle |\!\rangle, \\
& \langle\!| a! == 2, c! == \langle 1,0 \rangle |\!\rangle \}
\end{aligned}
$$

We find we need to include the output retrieve in each operation proof, and do a new output finalisation proof, in order to demonstrate refinement (see section 7).

We need to be a little careful about interpreting outputs when we have a non-trivial finalisation. It is not the raw output itself (a bitstream in the above example) that is observed, but rather that output *as viewed through finalisation.* So when interpreting the above specification, we do not 'observe' $\langle 0,0 \rangle$, we actually 'observe' *red*. In this example, this does not cause too much of a problem, because there is a bijection between each of the global, abstract and concrete outputs. However, it is possible to write specifications that 'confuse' or 'merge' apparently different abstract (or concrete) outputs to the *same* global one. For example, consider the case where the 'Parity' specification mentioned in section 6.2 is modified so that the abstract state (a sequence of bits) is apparently 'observed' by an abstract output comprising just this sequence, but where the abstract output is *finalised* to a global output comprising just the parity of the sequence. Then in fact only the parity has truly been observed (as can be demonstrated by refining the specification to a concrete one that outputs just

[3] by using Standard Z's [Z Standard 1995] explicit binding construction notation

the parity). So, when reading a specification the output finalisation has to be read along with the operation definition to understand precisely what is being observed.

Similar remarks apply to input initialisation.

In our own application, we performed input refinement. The abstract operation inputs a simple *go* to perform the transaction; the various steps in the concrete protocol input differing protocol messages.

6.4 State initialisation

With the Spivey rules, there is no Z-like state component in the global state, so state initialisation is independent of the global state. We relax this constraint and allow the global state to have a non-trivial Z-like component, and initialisation to depend on this value of the global state. This allows initial states to be set up with particular state values.

It is possible to model such initialisation by putting the system into some initial state unconstrained by the global state, then performing a state transition, using some special input, to 'initialise' it. But this is rather clumsy, and can pollute the model with a state component to capture whether the state has been initialised yet. More general initialisation permits cleaner models.

Also, having a specific initialisation allows the before global state to be related to the final global state, which means it is possible to express 'global-to-global' properties, for example, that a certain quantity is preserved.

In our own application, we found we needed to perform an initialisation based on the global state, because the actual devices entered the modelled world already having particular values for some of their components (for example, unique identification number).

6.5 Totalisation

The wider theory is cast in terms of *total* operations. When moving to a Z world, we have to decide how to treat Z's partial operations. We do this by choosing a totalisation: we assume the operation is chaotic outside its precondition, and moreover, that there is some 'bottom' state (representing a broken system), which is itself propagated appropriately chaotically to ensure that broken systems stay broken. (We require initialisation and finalisation to be total.)

Such a choice of operation totalisation corresponds to one common interpretation of the meaning of a Z operation outside its precondition: that 'anything can happen'. It also explains why refinement allows widening the precondition: it is simply a reduction of non-determinism, from completely chaotic to something rather less chaotic.

This chaotic totalisation corresponds to replacing a partial operation relation $op : X \longleftrightarrow X$ with its totalised counterpart $\overset{\bullet}{op} = (X^{\perp} \times X^{\perp}) \oplus op$, where $X^{\perp}$ is the set X augmented with an extra 'bottom' state. Other totalisations are possible, for example:

$$\overset{\bullet}{op} = (X^{\perp} \times \{\perp\}) \oplus op$$

$$\overset{\bullet}{op} = \mathrm{id}\, X^{\perp} \oplus op$$

Other totalisations correspond to other interpretations of the meaning of a Z operation outside its precondition, and result in different sets of refinement proof obligations. The most common is the 'firing condition' interpretation [Josephs 1991] [Strulo 1995], that *nothing* can happen outside the precondition.

So, the choice of a particular set of refinement proof obligations determines how a Z 'state and operations'-style specification should be interpreted.

In our own application, the traditional 'anything can happen' totalisation was appropriate for our model, so we did not need to derive more exotic rules.

7 Resulting data refinement proof obligations

The simplifying assumptions used to derive the Spivey rules from the relational refinement model are:

– The global relational state comprises only inputs and outputs. There is no Z-like state to initialise from, so state initialisation does not consider it, or finalise to, so state finalisation 'throws it all away'.
– The global, abstract, and concrete inputs and outputs are the same: the input initialisation, the output finalisation, and the respective retrieves, are the identity relation.
– Totalisation gives 'anything can happen' outside the precondition.
– The 'forward simulation' choice is made.

We have relaxed these assumptions in the following ways, to derive more widely applicable Z data refinement proof obligations:

– The global state can have a Z-like state component, and so the initial state can be related to the global one.
– State finalisation need not 'throw it all away'; state components can be observed without using outputs.
– Abstract and concrete inputs and outputs can differ, and are related by an appropriate retrieve; so i/o can be refined.
– Totalisation gives 'anything can happen' outside the precondition (as for the Spivey rules). The more widely applicable initialisation and finalisation steps are required to be total.
– Both the 'forward' and 'backward' choices are made.

These more relaxed choices lead to the more widely applicable Z data refinement proof obligations, given in sections 7.2 (forward rules) and 7.3 (backward rules).

7.1 Notation

In this section we introduce the various schemas used to define the refinement proof obligations in the following sections.

We use schemas to capture the global state G, global inputs $G_?$ and global outputs $G_!$. (We choose to bundle up our inputs and outputs each into a schema, to avoid polluting the description with their explicit types.)

The abstract model has abstract state A, inputs $A_?$ and outputs $A_!$. We initialise from the global to the abstract state with

$$AInit == [\, G; A' \mid \ldots]$$

We initialise from global inputs to abstract inputs with

$$AInit_? == [\, G_?; A_? \mid \ldots]$$

The abstract operation is

$$AOp == [\Delta A; A_?; A_! \mid \ldots]$$

and the abstract state finalisation and output finalisations are

$$AFin == [\, A; G' \mid \ldots]$$

$$AFin_! == [\, A_!; G_! \mid \ldots]$$

The concrete model has concrete state C, inputs $C_?$ and outputs $C_!$. The corresponding concrete initialisations, operation, and finalisations are

$$CInit == [\, G; C' \mid \ldots]$$

$$CInit_? == [\, G_?; C_? \mid \ldots]$$

$$COp == [\Delta C; C_?; C_! \mid \ldots]$$

$$CFin == [\, C; G' \mid \ldots]$$

$$CFin_! == [\, C_!; G_! \mid \ldots]$$

The concrete and abstract states, inputs and outputs are related by the retrieve relations:

$$R == [\, A; C \mid \ldots]$$

$$R_? == [\, A_?; C_? \mid \ldots]$$

$$R_! == [\, A_!; C_! \mid \ldots]$$

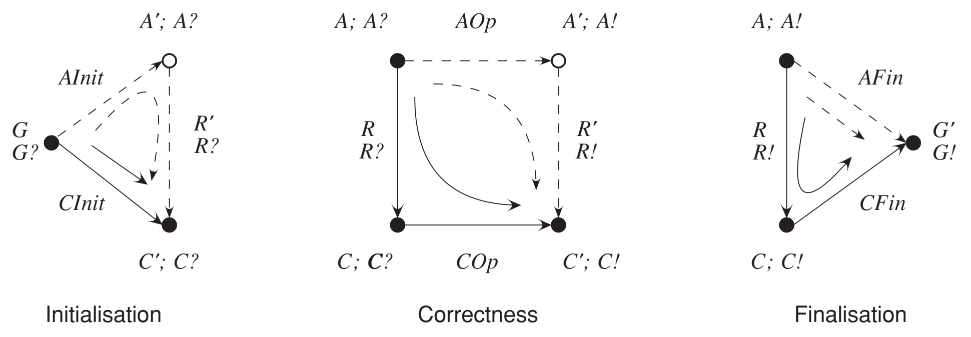

Fig. 8. The more powerful forward rules. The solid lines and states represent the hypothesised conditions; the dashed lines and circled states have to be proved to exist.

7.2 Forward rules

The forward rules are sketched in figure 8.

initialisation: for each concrete initial state obtained from some global state (and there must always be at least one), there must be a corresponding abstract initial state obtained from the same global state.

$$G \vdash \text{pre } CInit$$

$$CInit \vdash \exists\, A' \bullet AInit \land R'$$

For each concrete input obtained from some global input (and there must always be at least one), there must be a corresponding abstract input obtained from the same global input.

$$G \vdash \text{pre } CInit_?$$

$$CInit_? \vdash \exists\, A_? \bullet AInit_? \land R_?$$

applicability: whenever it is possible to perform the abstract operation AOp, it must be possible to perform the concrete operation COp on the corresponding concrete state and concrete input.

$$R;\ R_? \mid \text{pre } AOp \vdash \text{pre } COp$$

correctness: whenever it is possible to perform the abstract operation, and the corresponding concrete operation can result in state C' and output $C_!$, then it must be possible to find an abstract state A' and output $A_!$, corresponding to that C' and $C_!$, that is the result of performing the abstract operation.

$$R;\ R_?;\ COp \mid \text{pre } AOp \vdash \exists\, A';\ A_! \bullet AOp \land R' \land R_!$$

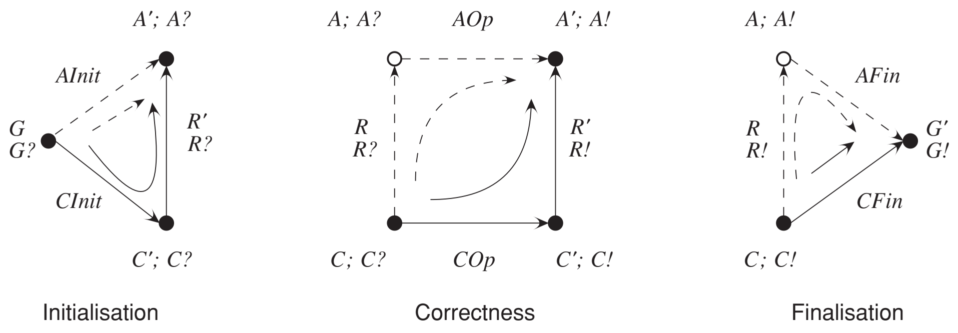

Fig. 9. The more powerful backward rules. The solid lines and states represent the hypothesised conditions; the dashed lines and circled states have to be proved to exist.

finalisation: for each concrete state that corresponds to some abstract state, where the concrete state finalises to global state G (and it must finalise), the corresponding abstract state must finalise to the same global state.

$R \vdash \text{pre } CFin$

$R; CFin \vdash AFin$

For each concrete output that corresponds to some abstract output, where the concrete output finalises to global output $G_!$ (and it must finalise), the corresponding abstract output must finalise to the same global output.

$R_! \vdash \text{pre } CFin_!$

$R_!; CFin_! \vdash AFin_!$

7.3 Backward rules

The backward rules are sketched in figure 9.

initialisation: for each concrete initial state obtained from some global state, that has a corresponding abstract state, that abstract state must be obtainable from the same global state.

$G \vdash \text{pre } CInit$

$CInit; R' \vdash AInit$

For each concrete input obtained from some global input, that has a corresponding abstract input, that abstract input must be obtainable from the same global input.

$G_? \vdash \text{pre } CInit_?$

$CInit_?; R_? \vdash AInit_?$

applicability: whenever it is possible to perform the abstract operation from all the abstract states and inputs corresponding to a chosen concrete state and input, then it must be possible to perform the concrete operation.

$$C;\ C_? \mid (\forall A;\ A_? \mid R \land R_? \bullet \text{pre } AOp\,) \vdash \text{pre } COp$$

correctness: for any abstract state A' corresponding to the after state C' reached from C by the concrete operation, there must be an abstract state A that both corresponds to C and is related to A' by the abstract operation.

$$\begin{array}{l} C;\ C_? \mid (\forall A;\ A_? \mid R \land R_? \bullet \text{pre } AOp\,) \\ \qquad \vdash \forall R';\ R_! \mid COp \bullet \exists A;\ A_? \bullet R \land R_? \land AOp \end{array}$$

When AOp is a *total* operation, this obligation reduces to

$$COp;\ R';\ R_! \vdash \exists A;\ A_? \bullet R \land R_? \land AOp$$

finalisation: for each concrete state that finalises to a global state G, there must be a corresponding abstract state that finalises to the same global state.

$$R \vdash \text{pre } CFin$$

$$CFin \vdash \exists A \bullet R \land AFin$$

For each concrete output that finalises to a global output $G_!$, there must be a corresponding abstract output that finalises to the same global output.

$$R_! \vdash \text{pre } CFin_!$$

$$CFin_! \vdash \exists A_! \bullet R_! \land AFin_!$$

8 Summary

We have been able to prove the correctness of the refinement of a real industrial product, working to real development time scales. In the process, we uncovered a security flaw in one part of the system design, and identified the corrections needed.

In the process of showing that it is possible to do Z refinement on an industrial scale and in an industrial context, we have learnt some lessons about how such a development can usefully be carried out. These lessons centre around the tension between two forces: the force driving one toward greater mathematical and aesthetic precision; and the other the desire to 'just do it' and achieve something useful for the project.

It is not possible to say that it is always better to follow the line of formality, or to say that it is always better to be strictly pragmatic. In some cases one decision must be made, and in other cases another. The guiding principle is usually the question of addressing the weakest link. Asking the question: "If I had only a week to complete this project, will I give greater assurance of correctness if I get this part more mathematically justified, or if I do proofs for more of the system, or if I present this more neatly, or ..."

A realistic assessment of this question will sometimes force you to get the maths right (no assurance is achieved if you don't know that your proof techniques work) and will sometimes force you to use the inelegant tools you have (no assurance is achieved if you prove nothing about the actual system in hand).

We achieved a very high level of rigour in our proofs. The proofs are far more detailed than typical proofs done in general mathematics. Despite this the formal methods activity was never on the critical path of the development. The formal methods component was usually ahead of schedule, and never caused a delay in development.

The success of this project has lead the client to do formal development, at the same level of rigour, on further products they are developing.

The proofs are also built on very sound theory: we investigated the foundations of the proof rules in great detail.

As a byproduct of doing these proofs, we have also improved the foundations of Z refinement rules. We have explained how the traditional Z data refinement proof obligations are the result of making certain simplifying assumptions, and embedding the Z world of state, inputs and outputs in a relational model. Different assumptions lead to different refinement rules; for our own application, we had to relax certain assumptions that go into deriving the Spivey rules in order to prove that our particular concrete specification was indeed a refinement on the abstract. We have presented here the actual proof obligations we discharged in our own application.

Other teams may well find they have to relax different assumptions, provide different computational models, or different totalisation embeddings, in order to prove their own refinements. ([Cooper *et al.*] will provide a detailed description of the precise derivations, in order that others may derive their own appropriate refinement proof obligations.)

Acknowledgements

The work described in the paper took place as part of a development funded by the NatWest Development Team.

Part of the refinement work was carried out by Eoin Mc Donnell.

References

[Cooper *et al.*]
David Cooper, Susan Stepney, and Jim Woodcock. *Refinement: Theory and Practice.* (in preparation).

[Flynn *et al.* 1990]
Mike Flynn, Tim Hoverd, and David Brazier. Formaliser—an interactive support tool for Z. In John E. Nicholls, editor, *Z User Workshop: Proceedings of the 4th Annual Z User Meeting, Oxford 1989*, Workshops in Computing, pages 128–141. Springer Verlag, 1990.

[He *et al.* 1986]
He Jifeng, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined (resumé). In *ESOP'86*, number 213 in Lecture Notes in Computer Science, pages 187–196. Springer Verlag, 1986.

[Josephs 1991]
Mark B. Josephs. Specifying reactive systems in Z. Technical Report TR-19-91, Programming Research Group, Oxford University Computing Laboratory, 1991.

[King & Arthan 1996]
D.J. King and R.D. Arthan. Development of Practical Verification Tools. *The ICL Systems Journal*, 11(1), May 1996.

[Meisels & Saaltink 1997]
Irwin Meisels and Mark Saaltink. *The Z/EVES Reference Manual.* 267 Richmond Road, Suite 100, Ottawa, Ontario, K1Z 6X3, Canada, June 1997. TR-97-5493-03c, http://www.ora.on.ca/z-eves/.

[Spivey 1992a]
J. Michael Spivey. *The ƒUZZ Manual.* Computer Science Consultancy, 2nd edition, 1992. ftp://ftp.comlab.ox.ac.uk/pub/Zforum/fuzz.

[Spivey 1992b]
J. Michael Spivey. *The Z Notation: a Reference Manual.* Prentice Hall, 2nd edition, 1992.

[Stepney]
Susan Stepney. Formaliser Home Page. http://public.logica.com/~formaliser/.

[Strulo 1995]
Ben Strulo. How firing conditions help inheritance. In Jonathan P. Bowen and Michael G. Hinchey, editors, *ZUM'95: 9th International Conference of Z Users, Limerick 1995*, Lecture Notes in Computer Science. Springer Verlag, 1995.

[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.

[Woodcock & Davies 1996]
Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, 1996.

[Z Standard 1995]
Z Notation version 1.2. Committee Draft Standard: CD13568. ISO panel JTC1/SC22/WG19, BSI panel IST/5/-/19/2, September 1995. http://www.comlab.ox.ac.uk/oucl/groups/zstandards/.