Methods and Tools for the Verification of Critical Properties

Roger Bishop Jones
International Computers Limited,
Eskdale Road, Winnersh, Berks, England, RG11 5TT.
Phone: +44 734 693131, E-mail: R.B.Jones@win0109.wins.icl.co.uk

22 July 2011

Abstract

This paper discusses methods for the formal treatment of critical systems. The discussion is based on experience at ICL in the application of formal methods to the development of highly assured secure systems.

Problems arising in the use of the standard paradigm for specification and refinement in Z are identified and discussed. Alternative methods which overcome some of these difficulties are presented.

A fully worked example is provided showing how the ICL ProofPower proof support tool may be used to specify and verify the critical properties of a secure system using the Z specification language.

The paper argues that effective use of formal methods in establishing, with high levels of assurance, that critical systems meet their critical requirements demands methods distinct from those typically advocated for general applications.

1 Introduction

This paper is an updated version of the paper of the same name presented by invitation at the 5th BCS FACS refinement workshop held in London in January 1992 [7]. The main changes are simply to update the proofs to work with ProofPower. Some additional small proofs have been included which illustrate reasoning about schemas as operations, which was not fully supported in the prototype. The arithmetic lemmas assumed as axioms in the previous proof have now been proven in the paper.

section 2 - methodological discussions

The primary methodological issues illustrated by the examples in the paper are introduced.

section 3 - ProofPower proof support for Z

This paper illustrates the use of ProofPower with Z. ProofPower has been used to produce the paper, to syntax and type check the specifications contained in it and in the development and checking of the formal proofs. A brief description of ProofPower is given together with information on how the various formal sections of the document are to be interpreted. These formal sections include not only Z specifications but also proof scripts for input to the proof tool and output from the proof tool as it processes these scripts.

section 4 - a formal model of a secure system

The paper begins with a presentation of a specification of a secure system in a fairly traditional way as a collection of Z schemas describing the operations of the system. This serves to illustrate some of the difficulties which motivate the alternative methods which form the main subject of the paper.

The primary difficulty identified at this stage is that specification of *a* secure system does not settle the question of what it is for a system to be secure, and to be able to verify that a system is secure one needs in the first instance a formal definition of *security*, not merely an example of a secure system.

section 5 - a specification of critical requirements

Having identified a need for a specification of critical requirement, an example of such a specification is furnished.

section 6 - a formal model of a system architecture

The first stage in the design and implementation of a system meeting the formalised critical requirements is to establish a system architecture which can be shown to guarantee satisfaction of the requirements. Such an architecture is exhibited, followed by a proof that it does indeed ensure conformance to the critical requirements.

section 7 - a model of a secure kernel

The correctness proposition for the architectural design asserts that a system correctly constructed from subsystems having certain critical properties will meet the critical requirements for the system. We now proceed to exhibit a (mathematical model of a) subsystem having the required critical property. The proof that the kernel does guarantee that the system is secure is shown in full.

section 8 - system correctness proposition

For completeness the overall system is defined and its correctness proposition stated.

2 Methodological Discussions

For over 5 years ICL has been developing and delivering to external customers formal machine checked proofs about secure systems. This paper attempts to identify and illustrate some of the special concerns that arise in the use of formal methods for the development of very high assurance systems. No deep insights are reflected in the methods we have adopted, which flow naturally from a preoccupation with properties of systems. Nevertheless, we are aware of few examples of published material on this topic. Rob Arthan presents similar methods applied to the specification of the critical properties of a proof tool in the proceeding of VDM 91 [1]. Jeremy Jacob discusses some of the problems in refining specifications of secure systems in [6].

The main methodological issues raised in this paper are:

- The need to verify critical systems against a specification of critical requirements.
- The form of specifications, and the distinction between a specification of critical requirements and a full functional specification.
- The traditional notion of refinement and its relationship with the meaning of specifications.
- The limitations of the traditional notion of specification for expressing critical requirements.
- The need for an explicit formalisation of the claim to be verified.

I propose to focus primarily upon two recommendations.

2.1 Formalisation of Critical Requirements

The first recommendation is that formal treatments of critical systems should begin with formalisation of *critical properties* and that the primary objective of the formal treatment should be to establish that the system as implemented will have these properties.

This contrasts with the more common perception that the formal treatment begins with an abstract formal specification of the system and that the primary objective of the formal treatment is to establish that the system as implemented is "correct" with respect to this initial specification. Often the desirability of proving critical properties of specifications is mentioned as a way of checking the correctness of a system specification, without any prior mention of the need to formalise these properties, as if these properties are in general straightforward and obvious enough to deserve no attention until we feel it necessary to prove them.

We advocate that in the case of *critical* systems (whether security or safety critical) the specification of critical properties should be regarded as the most important aspect of the formal treatment, since these represent the most crucial aspects of the requirements on the system being developed. There is no reason to suppose that either a specification or a design can be undertaken successfully if these critical requirements have not first been established.

The connection implicit in this discussion between critical *requirements* and critical *properties* is important. Critical requirements cannot in general be adequately expressed other than by stating a *property* of systems, which is not the same thing as a *model* of *a system*. Safety and security are properties not models.

2.2 Caution about Refinement

My second recommendation is that care should be taken to ensure that the models of the system which are used in this process are clearly understood, and are adequate for the purposes in hand.

Particularly the reader is warned to beware of the use of models which are intended to permit "operation refinement" (as defined, for example, in *The Z Notation* [10] section 5.5). The reason why he should beware of such models is that properties possessed by such models need not be enjoyed by "correct" refinements of them under the normal rules of refinement. It is therefore not adequate to write a specification in Z of a system, prove that this specification has the required critical properties, and then refine to an implementation using the normal rules of operation refinement. Similar considerations also apply to data refinement.

This does not mean that such models may not be used. It means that proof of critical properties cannot in general be mediated by a proof that an incomplete specification of the system has the required properties.

The reason why these problems arise is that a model of this kind is by convention regarded as not specifying the possible inputs to the system. So called "pre-conditions" are interpreted not as constraining the possible inputs to the system, but as constraining the scope of applicability of the constraints imposed by the post-conditions and state-invariants. The system specifier wishing to ensure that, whatever the behaviour of the operator, a system will not behave dangerously, will find that a legitimate refinement of his system may introduce further possible inputs, and that the behaviour of the system when receiving these inputs is in no way constrained by any invariants or post-conditions which the original specifications contained.

The presumption that systems can be shown to be secure by showing that a model of the system is secure, and then proving that the system is correctly refined from the model has never featured in the methods used by ICL for high assurance secure systems. This kind of problem in the development of secure systems has been discussed in the literature by Jeremy Jacob [6].

The kind of properties of models which are preserved under refinement have been called "safety properties". This unfortunate terminology might suggest to the reader that the problems discused here arise only in the development of secure systems, and not in the development of safety critical systems. There is however no reason to suppose that the critical properties required of safety critical systems are "safety properties" in this technical sense. Using the term "reckless refinement" to describe refinements which permit liberalisation of the pre-condition without preservation of state invariants or post-conditions, then "safety properties" are those which are indifferent to reckless refinement. It is doubtful that any safety critical systems have critical properties which are indifferent to reckless refinement.

Reckless refinement is like permitting an extra cockpit control to be introduced which overrides all the safety features in the flight control software.

3 ProofPower Support for Z

3.1 Background

The ICL Secure Systems High Assurance Team has grown from the need to use formal methods in the development of highly assured secure systems.

The dominant formal specification notation in the UK for such applications is Z[9], and the evaluation guidelines for secure systems call not only for formal specifications, but also for proofs.

At the time when ICL was establishing its capability in this area the Z notation was less stable and well defined than it now is, and lacked good tool support. We nevertheless wanted to obtain early experience of undertaking formal proof and therefore spent some time looking at proof support tools for languages other than Z. Some experience was obtained in the use of the NQTHM theorem prover (often called the Boyer-Moore prover after its authors [2]), and also of the HOL system developed at the University of Cambridge by Mike Gordon and others [3][4].

The HOL system was felt to be more suitable for our applications for two main reasons.

The first reason was that the language supported by the system, a polymorphic version of Church's simple theory of types (using Milner-style type polymorphism [8]), was closer in its logical expressiveness to Z than the quantifier-free first order logic supported by NQTHM.

The second factor was that the HOL system was one of the LCF family [5]. This meant that the role of the user of the tool in the proof development process was more fully recognised, and a powerful meta-language (ML) was available for the user to use in programming proofs. The flexibility of the system for adaptation to tasks not anticipated by the developers of the system was felt to be greater than that of NQTHM.

We found that HOL as a language was close enough to Z that the kind of specifications we were then writing could be manually translated into HOL without great difficulty. This provided the beginnings of our practical experience in constructing formal machine checked proofs. The flexibility and extendibility afforded by the LCF paradigm enabled us to progressively customise the HOL system towards support for reasoning about specifications translated from Z, and we were able to acquire gradually a deeper understanding of these languages and of the relationship between them.

By mid 1988 we were sufficiently convinced of the merits of the HOL system that when an opportunity arose to undertake a technology research and development project our proposal was based around an industrial re-implementation of the HOL system. One of our objectives was to achieve best possible proof support for Z, but at the time of submitting the proposal we were uncertain about how close that would be, and our proposal was therefore non-committal. The language had by this time been given a formal semantics (although incomplete) by Spivey [9], but there was no sign of progress on proof rules. Compared with the stark simplicity of HOL, Z seemed a very complex language, and contained some features which seemed logically controversial. One of particular concern was the fact that variables in the signatures of schemas effectively occur in expressions in which they do not syntactically occur. The rules for free and bound occurrences of variables would either have to be very unusual, or else these concepts would not be adequate to express the necessary side conditions on the logical rules. Other oddities include the lack of distinction between variables and constants (which makes side conditions non-local).

We proposed to continue to support proof for Z via HOL. We felt that a re-implementation of the HOL system following industrial quality control processes was desirable and that this would give us valuable further experience with the technology to enable us to provide best achievable support for Z. Early into the project further investigations led us to conclude that full support for the Z language as then defined was feasible by this route, and since then this has been one of the primary objectives of our work on proof support.

3.2 ProofPower Z Proof Support

The ProofPower Z proof support facilities are sufficiently powerful to permit illustration of the methods described in this paper.

We have not felt that our objectives in relation to proof support could be achieved if substantial improvements in support for the preparation of Z scripts were attempted at the same time, and therefore the preparation of scripts is undertaken using standard text editors, though benefiting from the use of extended fonts. The development method is document based, where the documents are LaTeX source scripts interspersed with formal text in a "close to wysiwyg" format. This paper is such a script.

During the preparation of a script the developer may have running concurrently with the text editor an interactive session of ProofPower, which will undertake the syntax and type checking of specifications in an incremental manner, and will support interactive development of proof scripts.

In the case of specification checking this may be achieved interactively by the use of cut-and-paste from the editor into the proof tool. As the specification is entered and checked, theories are built up containing the formal specifications. Proofs of conjectures relating to the specifications may be undertaken at any time, and the resulting theorems, once proven, may be stored in theories with the specifications.

The entire process may be rerun in batch mode to ensure that the resulting document is complete and correct.

The proof system is based on the LCF paradigm, under which proof scripts are essentially programs in standard ML which compute the required theorems via abstract data types which ensure that all computations of theorems correspond to admissible inferences in the supported logic. Proof scripts of this kind are not usually intelligible without seeing the intermediate goals displayed by the proof tool during the development of the proof. If proof scripts are intended to be read without resort to the tool they are therefore best annotated with output from the proof tool showing these intermediate goals.

This document is a "literate script" containing, in addition to informal narrative, formal specifications and annotated proof scripts. Specifications are all in the language "Z". They are preceded by a small Z on the left of the page, and where not enclosed by a box according to the Z conventions, a vertical bar on the left makes clear the extent of the formal material.

The proof script proper is in the language ML (in fact "standard" ML), and is marked by vertical bars on the left starting with the characters "ML". Output from the proof tool has been included in the document, again distinguished by a vertical bar on the left, in this case headed by 'ProofPower output'. Descriptions of some of the proof facilities available are included in the narrative. The output displayed is not quite verbatim. To improve the readability of the paper I have added newlines to overcome shortcomings of the pretty printer in the prototype. Some of the duplication of output which is beneficial in an interactive session using a scrolling teletype interface has been eliminated, "..." marking the place where non-current subgoals, or irrelevant assumptions have been listed.

The detailed formal scripts presented demonstrate how our prototype tool provides assistance in finding proofs as well as in checking them. It illustrates some of the additional complexity arising in Z proofs relative to proofs in HOL, and some of the mechanisms so far developed for dealing with these complications.

4 A Formal Model of a Secure System

The use of formal techniques in systems developments may be beneficial even if no proofs are undertaken.

Our concern in this paper however, is not with the use of formal techniques in this (rather informal) manner. We are concerned with formal techniques which are appropriate where proofs are to be undertaken to give higher levels of assurance that the system under development will have certain critical properties, such as "security" or "safety".

In such cases, it is our belief that a prerequisite of obtaining value from undertaking proof is to obtain some *formal* statement of what is to be proven.

In order to make effective use of formal techniques to obtain high assurance it is necessary to establish formally the proposition of which we seek assurance.

This may seem to be an obvious requirement, but it is a requirement which is not supported by standard methods. In the case of 'model oriented' specifications in Z there is in general no single formal entity which represents the system as a whole. In 'model oriented' specification methods the conventional paradigm is to have (or to hope for) a tool which will generate proof obligations. What is proven is a set of propositions (perhaps generated by this tool), which are expected to amount to a correctness result, even though the proposition which these proof obligations establish is not itself expressed or proven by the proof system.

4.1 Informal Description of Security

"Security" is meant , for the purposes of this paper in the narrow sense of preserving the confidentiality of classified information. A secure system is one which stores data classified according to its 'sensitivity'. 'sensitivity' is a measure of how serious unauthorised disclosure of the information is considered to be.

Sensitivity classifications are normally (partially) ordered, and in the following examples they will be modelled by natural numbers. Every user of the system has a clearance, which is also a natural number. The system is required to permit a user to access only data whose classification is not greater than the user's clearance.

Transfer of information from an entity of a certain classification to an entity having a lower classification is known as a downgrade. The system is required not to connive in the downgrade of information. This means that the system itself will not undertake a transfer of information which might reduce the classification of the information.

The user himself may be entitled both to read highly classified data, and to write to more lowly classified data. If so he may take notes himself from highly classified data and use the information obtained while modifying lowly classified data. In doing so he will be committing a breach of security. The system will not be able to prevent such breaches, but it will prevent all breaches which it is able to detect. In particular the user will not be permitted to copy a highly classified object into a lowly classified object by instructions to the system.

4.2 The Model

The following example is presented as an example of a specification in the Z model-oriented style of a system purporting to meet the requirements informally described in the previous section. It is not offered as a realistic specification of a secure system. Its primary purposes are:

- to clarify by formal example the informal notion of secure system
- to illustrate some of the points subsequently raised about the role of this kind of specification in the development of critical systems

The proof tool keeps specifications in theories and we therefore introduce a new theory for the specification to follow:

```
SML
set_flag("z_type_check_only", false);
set_flag("z_use_axioms", true);
open_theory"z_library";
set_pc "z_library";
new_theory"wrk050";
new_parent(hd (get_cache_theories()));
```

We have no interest in the form of data to be stored by our computer system, and therefore introduce a "given set" for this data.

Z

$[\mathbf{DATA}]$

We do however require that the system stores data together with a *classification.* For simplicity we assume that all data at a given class is stored together indexed by its classification, and we use non-negative numeric values as classification marks.

Z

STATE

$classified\_data : \mathbb{N} \rightarrow\!\!\!\!\!\!+ \, DATA$

Z

OPERATION

$\Delta STATE;$

$\mathbf{clear}? : \mathbb{N}$

z
READ

$$
\begin{array}{l}
\mathit{OPERATION};\\
\mathbf{class}? :\mathbb{N};\\
\mathbf{data}! \ : \mathit{DATA}\\
\hline
\mathit{class}? \in \mathit{dom}\ \mathit{classified\_data};\\
\mathit{class}? \leq \mathit{clear}?;\\
\mathit{data}! = \mathit{classified\_data}\ \mathit{class}?;\\
\mathit{classified\_data}' = \mathit{classified\_data}
\end{array}
$$

z
COMPUTE

$$
\begin{array}{l}
\mathit{OPERATION};\\
\mathbf{class}? :\mathbb{N};\\
\mathbf{computation}? \quad :(\mathbb{N} \nrightarrow \mathit{DATA}) \rightarrow \mathit{DATA}\\
\hline
\mathit{class}? \in \mathit{dom}\ \mathit{classified\_data};\\
\mathit{class}? \geq \mathit{clear}?;\\
\mathit{classified\_data}'\\
= \quad \mathit{classified\_data} \oplus\\
\qquad \{\mathit{class}? \mapsto \mathit{computation}?\ ((\mathit{0} \mathrel{..} \mathit{clear}?) \lhd \mathit{classified\_data})\}
\end{array}
$$

In this case the clause in the pre-condition of the operation which is necessary to ensure that the operation is secure is $\mathit{class}?\geq\mathit{clear}?$. The difference between the two operations in this area is because *class?* in this operation is a destination for information not a source. This clause does not suffice, but is supplemented by the fact that the state is filtered of highly classified data before being used in the computation. This ensures that the user is not permitted access to information for which he is not cleared.

These two operations are supplied as representatives of secure operations. Examples of insecure operations may be obtained by omitting the constraint on *class*? in either of the above operations.

4.3 Discussion of Specification

4.3.1 Specification of Systems or of Properties of Systems

In the previous section we have supplied a specification of a simple system. The state of the system contains a classified data store. Two operations have been specified which we believe to be “secure” in a sense characterised informally in the narrative.

It is clear that even in this simple specification minor errors might have resulted in the specified system not meeting our informal notion of security. In fact an earlier version of this very simple example survived a week before I realised that it was not secure. In a realistic system specification, which would be considerably more complex, the risk of the system described being insecure *because of errors in the specification* becomes significant. No amount of care or proof in designing and implementing a system from such a specification would result in the system being secure.

We would like to be able to use formal techniques to establish whether a system specified in this (or some other way) is secure. For this purpose a specification of a secure system is not sufficient. What is needed is a specification of the property of being secure, only then are we in a position to judge whether any specified system is secure.

The reader might object that the problem facing us is one of infinite regress. No matter where we start our formal work there is a risk that errors will be made, and these errors will not be formally checkable against some previous specification. Nevertheless, it is our experience that a specification of what it is to be secure (of our *critical property*) is much simpler than the complete specification of some particular secure system. The activity of formalising the critical requirement is one in which the specifier focuses exclusively on the aspects of the system of highest concern. The informal assessment of the critical requirements is likely to be more effective because of this focus, and because of the *smaller* specifications which result.

4.3.2 Further Problems

Even as a specification of a system, what we have offered cannot be accepted, though superficially plausible, as a specification of a secure system.

This is because the standard rules of refinement, which fill an important gap between the formal meaning of the specification and its *true* meaning, will permit this specification to be refined by liberalisation of the pre-condition into specifications which are no longer even superficially those of secure systems.

To ensure that this cannot formally be refined into an insecure system it is necessary to specify what happens when the preconditions are not met. This is exacerbated in Z by the fact that the formal precondition is not what it might appear to be. Thus the formal precondition includes the state invariant on the initial state (you may not assume that the system states satisfy the invariant), and also includes the predicate implicit in the declaration of the input variables. The specification can legitimately be refined to one in which *clear?* is a negative integer, and the behaviour of the system in such circumstances is not constrained unless a schema is supplied in which this is not explicitly or implicitly in the precondition.

These considerations are important in the treatment of critical systems, and may be the origin of the belief among some that formal methods (in general) can specify what a system must do, but cannot specify what it must not do.

My list is not exhausted. A further difficulty in some applications (including security) is that this form of specification confuses looseness and non-determinism. A non-deterministic system is one which does not always exhibit the same behaviour when supplied with the same inputs and initial state. Non-deterministic systems are particularly problematic when confidentiality is at issue. There are special problems in deciding when information flows are occurring in non-deterministic systems, so that there is as yet no consensus on what it means to say that a system is secure (in the narrow sense of enforcing confidentiality) if it is non-deterministic. The formalisation of security given below is essentially a property of deterministic systems, and so cannot be applied to a system model which may be interpreted as or refined to a non-deterministic system.

If we were to attempt to formalise the notion of information flow security for non-deterministic systems, then it is unlikely that the kind of model of a non-deterministic system provided by a Z schema would be satisfactory. In order to make statements about information flow through non-deterministic systems it is necessary to know not only the possible outcomes of an operation, but also the probability distribution. Even a very small amount of noise on a channel may render all transitions possible without significantly impairing the ability of the channel to transmit information.

So a Z schema, interpreted as a specification of an operation, not only fails to provide the means of specifying deterministic systems, but provides models of non-deterministic systems which may not be adequate for some purposes. This is not to be construed as a criticism of the language Z, which is a rich notation for classical set theory well able to cope with any mathematical modelling task. It is a criticism of the narrow view that systems should or can be modelled by Z schemas, and the normal interpretation of such models.

4.4 Some Proofs

The following proofs serve to illustrate:

1. The use of ProofPower for reasoning about this kind of specification.
2. That security need not be preserved by "correct" refinements.

The first two proofs show what the pre-conditions of OPERATION and READ are. As presented it appears that the user must make a conjecture about the precondition before commencing the proof. In practice this is not the case. In both cases below the pre-condition was established using an exploratory attempt at proving the propositions ⓩ*pre OPERATION* ⇔ *true*⌝and ⓩ*pre READ* ⇔ *true*⌝. If these propositions are not true then they will reduce during proof to subgoals which entail the relevant pre-condition (and, provided that care is taken with strengthening tactics, the conjunction of the subgoals will actually be equivalent to the pre-condition). If the original goal was not true these proofs cannot be completed, but they provide the necessary information to set up and prove the correct statement of the pre-conditions. The final proofs are very similar to the exploratory proofs, so little additional effort is involved. A tool could be constructed which would enable the pre-condition to be simplified without adopting this two stage approach, but this would be at the cost of disallowing steps which are not known to be equivalence transformations, which would offset the useability advantages obtainable by this more direct approach.

First we demonstrate the pre-condition of OPERATION:

SML
```
set_goal ([], ⓩpre OPERATION ⇔ classified_data ∈ ℕ ⇸ DATA ∧ 0 ≤ clear?⌝);
```

The first step in the prove is to expand the schemas involved by rewriting.

SML
```
a (rewrite_tac (map z_get_spec [ⓩOPERATION⌝, ⓩSTATE⌝]));
```

ProofPower output
```
...
(* ?⊢ *)  ⓩ(∃ classified_data′ : 𝕌
                • (classified_data ∈ ℕ ⇸ DATA
                  ∧ classified_data′ ∈ ℕ ⇸ DATA)
                  ∧ 0 ≤ clear?)
              ⇔ classified_data ∈ ℕ ⇸ DATA ∧ 0 ≤ clear?⌝
...
```

This subgoal can be simplified by stripping and then rewriting the resulting conclusions with the available assumptions.

SML

```
a (REPEAT z_strip_tac THEN_TRY asm_rewrite_tac[]);
```

ProofPower output

```
...
(∗  2 ∗)  ⓩclassified_data ∈ ℕ ⇸ DATA⌝
(∗  1 ∗)  ⓩ0 ≤ clear?⌝

(∗ ?⊢ ∗)  ⓩ∃ classified_data′ : 𝕌 • classified_data′ ∈ ℕ ⇸ DATA⌝

...
```

We now have an existential subgoal which is clearly true, and can be proven so by furnishing the most trivial witness and rewriting in an extensional proof context containing sufficient knowledge of the Z ToolKit. The use of *PC_T1* "*z_library_ext*"causes a local switch into the proof context *z_library_ext*.

SML

```
a (z_∃_tac ⓩ{}⌝ THEN PC_T1 "z_library_ext" rewrite_tac[]);
```

ProofPower output

```
Tactic produced 0 subgoals:
Current and main goal achieved
...
```

SML

```
save_pop_thm "preOP";
```

Demonstrating the pre-condition of READ is more complicated but introduces no new problems.

SML

```
set_goal ([], ⓩpre READ ⇔
                classified_data ∈ ℕ ⇸ DATA
                ∧ 0 ≤ clear?
                ∧ 0 ≤ class?
                ∧ class? ∈ dom classified_data
                ∧ class? ≤ clear?⌝);
a (rewrite_tac (map z_get_spec [ⓩREAD⌝, ⓩOPERATION⌝, ⓩSTATE⌝])
        THEN REPEAT z_strip_tac THEN_TRY asm_rewrite_tac[]);
```

ProofPower output

```
...
(*  5 *)  ⓩclassified_data ∈ ℕ ⇸ DATA⌝
(*  4 *)  ⓩ0 ≤ clear?⌝
(*  3 *)  ⓩ0 ≤ class?⌝
(*  2 *)  ⓩclass? ∈ dom classified_data⌝
(*  1 *)  ⓩclass? ≤ clear?⌝

(* ?⊢ *)  ⓩ∃ classified_data′ : 𝕌; data! : 𝕌
              • (classified_data′ ∈ ℕ ⇸ DATA
                ∧ data! ∈ DATA)
                ∧ data! = classified_data class?
                ∧ classified_data′ = classified_data⌝
..
```

SML

```
a (z_∃_tac ⓩ(classified_data′ ≙ classified_data, data! ≙ classified_data class?)⌝
        THEN asm_rewrite_tac [z_get_spec ⓩDATA⌝]);
```

ProofPower output

```
Tactic produced 0 subgoals:
Current and main goal achieved
```

SML

```
save_pop_thm "preREAD";
```

Now we define BADREAD, which is intended to be a refinement of READ:

Z

```
┌─BADREAD──────────────────────────────────
│    OPERATION;
│    class? :ℕ;
│    data!  : DATA
├──────────────────────
│    READ ∨
│    (class? > clear?;
│    data! = classified_data class?;
│    classified_data′ = classified_data)
└──────────────────────────────────────────
```

Now we prove that BADREAD is a correct refinement of READ. The approach to the proof is very similar to that of the pre-condition proofs.

rename_tac is invoked between stripping and rewriting with the assumptions in this case because otherwise the rewriting fails. In some cases occurrences of logically distinct variables with the same name interferes with rewriting in ways which could not be avoided without unacceptable performance overhead. In such cases rewriting may fail altogether, or may simply fail to do as much as could be done. *rename_tac* changes the variable names to reduce this problem.

SML

```
set_goal([], ⓩ(pre READ ⇒ pre BADREAD) ∧ (pre READ ∧ BADREAD ⇒ READ)⌝);
a (rewrite_tac (map z_get_spec [ⓩBADREAD⌝, ⓩREAD⌝, ⓩOPERATION⌝, ⓩSTATE⌝]));
a (REPEAT z_strip_tac THEN rename_tac[] THEN asm_rewrite_tac[]);
```

ProofPower output

```
...
(* 9 *)  ⓩclassified_data ∈ ℕ ⇸ DATA⌝
(* 8 *)  ⓩclassified_data′ ∈ ℕ ⇸ DATA⌝
(* 7 *)  ⓩ0 ≤ clear?⌝
(* 6 *)  ⓩ0 ≤ class?⌝
(* 5 *)  ⓩdata! ∈ DATA⌝
(* 4 *)  ⓩclass? ∈ dom classified_data⌝
(* 3 *)  ⓩclass? ≤ clear?⌝
(* 2 *)  ⓩdata! = classified_data class?⌝
(* 1 *)  ⓩclassified_data′ = classified_data⌝

(* ?⊢ *)  ⓩ∃ classified_data″ : 𝕌; data!′ : 𝕌
            • (classified_data″ ∈ ℕ ⇸ DATA
              ∧ data!′ ∈ DATA)
              ∧ ((classified_data″ ∈ ℕ ⇸ DATA
                  ∧ data!′ ∈ DATA)
                  ∧ data!′ = classified_data class?
                  ∧ classified_data″ = classified_data
                ∨ clear? < class?
                  ∧ data!′ = classified_data class?
                  ∧ classified_data″ = classified_data)⌝
...
```

SML

```
a (z_∃_tac ⓩ(classified_data″ ≙ classified_data, data!′ ≙ classified_data class?)⌝
        THEN asm_rewrite_tac[]);
```

ProofPower output

```
...
(* 9 *)  ⓩclassified_data ∈ ℕ ⇸ DATA⌝
...
(* 4 *)  ⓩclass? ∈ dom classified_data⌝
...
(* ?⊢ *)  ⓩclassified_data class? ∈ DATA
            ∧ (classified_data class? ∈ DATA ∨ clear? < class?)⌝
...
```

Here forward chaining using the useful facts about membership of applications in *z_fun_∈_clauses*,

```
|⊢ ∀ f : 𝕌; x : 𝕌; X : 𝕌; Y : 𝕌
|   • ((f ∈ X → Y ∨ f ∈ X ↣ Y ∨ f ∈ X ↠ Y ∨ f ∈ X ⤖ Y)
|       ∧ x ∈ X ⇒ f x ∈ Y)
|   ∧ ((f ∈ X ⇸ Y ∨ f ∈ X ⤔ Y ∨ f ∈ X ⤀ Y)
|       ∧ x ∈ dom f ⇒ f x ∈ Y) : THM
```

solves the problem.

SML

```
|a (all_fc_tac [z_fun_∈_clauses]
|        THEN REPEAT strip_tac);
```

ProofPower output

```
|Tactic produced 0 subgoals:
|Current and main goal achieved
|...
```

SML

```
|save_pop_thm "BADrefine";
```

5 A Specification of Critical Requirements

SML

```
|open_theory "wrk050";
```

Z

```
|        [IN,OUT]
```

The state of our system is similar to that in the previous example. Classifications of data and clearances of users are modelled by natural numbers, more sensitive data having numerically higher classifications.

Since there is only one component in the state it is simpler not to use a schema.

Z

```
|        STATE2 ≙ ℕ ⇸ DATA
```

A system is modelled as a transition function. This is to be regarded as a single function modelling all the permissible transitions of the system. We require that the system is a total function.

Z

```
|        SYSTEM ≙ (ℕ × IN × STATE2) → (STATE2 × OUT)
```

We now attempt to capture (in fact *define*) what it means to say that such a system is *secure*. The intended meaning here concerns the nature of the information flows permitted by the system. The requirement will be expressed by two properties concerning the information flowing to the output of the system, and the information flowing between different classes of data within the state.

The constraint on information flowing to the output is that none of this information comes from data classified more highly than the clearance associated with the input (which is to be understood as the clearance of some user supplying the input and receiving the output).

This is expressed by saying that the output will be the same if the state before the transition differed only in data classified more highly than the input.

z

$$
\begin{array}{|l}
\textbf{out\_secure} : \mathbb{P}\ \mathit{SYSTEM} \\
\hline
\forall \mathit{sys}{:}\mathit{SYSTEM} \bullet\ \mathit{sys} \in \mathit{out\_secure} \Leftrightarrow \\
(\forall\ \mathit{clear}{:}\mathbb{N};\ \mathit{inp}{:}\mathit{IN};\ s,s'{:}\mathit{STATE2} \\
\mid (0\ ..\ \mathit{clear}) \lhd s = (0\ ..\ \mathit{clear}) \lhd s' \\
\bullet \qquad \mathit{second}\ (\mathit{sys}\ (\mathit{clear},\ \mathit{inp},\ s)) \\
\qquad = \quad \mathit{second}\ (\mathit{sys}\ (\mathit{clear},\ \mathit{inp},\ s')))
\end{array}
$$

The constraint on flows within the state are that the information flowing to any data in the state should be derived exclusively from information in the state which is classified no higher than the destination data.

z

$$
\begin{array}{|l}
\textbf{state\_secure} : \mathbb{P}\ \mathit{SYSTEM} \\
\hline
\forall \mathit{sys}{:}\mathit{SYSTEM} \bullet\ \mathit{sys} \in \mathit{state\_secure} \Leftrightarrow \\
(\forall \mathit{class},\ \mathit{clear}{:}\mathbb{N};\ \mathit{inp}{:}\mathit{IN};\ s,s'{:}\mathit{STATE2} \\
\mid ((0\ ..\ \mathit{class}) \lhd s) = ((0\ ..\ \mathit{class}) \lhd s') \\
\bullet \qquad (0\ ..\ \mathit{class}) \lhd (\mathit{first}\ (\mathit{sys}\ (\mathit{clear},\ \mathit{inp},\ s))) \\
\qquad = \quad (0\ ..\ \mathit{class}) \lhd (\mathit{first}\ (\mathit{sys}\ (\mathit{clear},\ \mathit{inp},\ s'))))
\end{array}
$$

A further property involving the flows of information from the input value would also be appropriate, but is omitted for the sake of simplicity. The security property is then simply the conjunction of these two properties.

z

$$
\begin{array}{|l}
\textbf{secure} : \mathbb{P}\ \mathit{SYSTEM} \\
\hline
\forall \mathit{sys}{:}\mathit{SYSTEM} \bullet\ \mathit{sys} \in \mathit{secure} \Leftrightarrow \mathit{sys} \in \mathit{state\_secure} \\
\qquad\qquad\qquad\qquad\qquad \land\ \mathit{sys} \in \mathit{out\_secure}
\end{array}
$$

6 A Formal Model of a Secure System Architecture

The system may be implemented minimising the critical function, by implementing a secure *KERNEL*, which is required to enforce security, and an untrusted *APPLICATION*.

The kernel is a subsystem which will control the running of the application, giving access to the data store at a level which is appropriate to the clearance of the current user of the system. When a command is initiated the system is supplied with the clearance of the user. This information is used by the kernel to control access by the application to the data store.

6.1 Component Types

The application is modelled as if it were a system itself, except that it need not concern itself with security, and is therefore not supplied with the clearance parameter associated with each operation.

z

$$\textbf{APPLICATION} \mathrel{\widehat{=}} (\mathit{IN} \times \mathit{STATE2}) \rightarrow (\mathit{STATE2} \times \mathit{OUT})$$

In general a security kernel will be capable of controlling an application because the processor on which it runs has hardware protection facilities which make this possible. More detailed and realistic models of this kind of system would therefore be expected to show how such features of the hardware supported the enforcement of security.

However, secure systems can be built using processors which do not have such protection features, at the cost either of disallowing assembly level implementations of applications and regarding the compiler as a critical subsystem, or of arranging for the application to be interpreted by the kernel rather than executed by the hardware.

The following model of a kernel is most plausible in these contexts. The application could be a functional program, provided to the kernel for invocation as appropriate, or any other sort of program to be interpreted by the kernel.

The kernel is modelled as a function which when supplied with an *APPLICATION* will yield a *SYSTEM*.

z

$$\textbf{KERNEL} \mathrel{\widehat{=}} \mathit{APPLICATION} \rightarrow \mathit{SYSTEM}$$

6.2 Construction

Given the way the kernel has been modelled, the construction of a system from a kernel and an application is trivial.

z

$$\textbf{construction} : \mathit{APPLICATION} \times \mathit{KERNEL} \rightarrow \mathit{SYSTEM}$$

$$\forall \mathit{appl}{:}\mathit{APPLICATION};\ \mathit{kernel}{:}\mathit{KERNEL} \bullet \\ \quad \mathit{construction}\ (\mathit{appl},\ \mathit{kernel}) = \mathit{kernel}\ \mathit{appl}$$

6.3 Critical Requirements on Components

The architecural thinking here goes little further than that there will be a kernel and it will be responsible for enforcing the security of the system. The critical requirements on the kernel can therefore say simply that the kernel must be capable of discharging this responsibility, i.e. that whatever application is supplied to the kernel, the resulting system will be secure.

```
Z
│    secure_kernel : ℙ KERNEL
├──────────────────────────────
│    ∀ kernel:KERNEL• kernel ∈ secure_kernel ⇔
│    (∀appl:APPLICATION• (construction (appl, kernel)) ∈ secure)
```

6.4 Architecture Correctness Proof

We are now in a position to formulate a conjecture expressing the claim that the architectural design modelled above suffices for the construction of secure systems.

The conjecture states that whenever a system is built using *construction* from a kernel which is a *secure_kernel* together with any application, then the resulting system will be *secure*. The fact that no conditions on the nature of the application are expressed indicates that the application need not be trusted to behave in any particular way. Whatever the behaviour of the application the system will be secure.

This claim is trivial and so is its proof, since it follows directly from the *definition* of a *secure_kernel*.

Nevertheless the machine proof will be exhibited, as our first introduction to the mechanics of proof.

Though proof checking in LCF-like systems (including ProofPower) is undertaken on forward inferences from axioms to theorems, the normal interactive proof style is a backward style supported by a "subgoal package". The subgoal package is responsible for translating the results of the goal oriented backward proof search into a fully checked forward proof.

```
SML
│set_pc "z_library";
```

```
SML
│val secure_kernel_sim = z_defn_simp_rule (z_get_spec⌜Z secure_kernel⌝);
```

The proof is initiated by first giving the conjecture to the goal package as follows:

```
SML
│set_goal([],⌜Z ∀kernel:KERNEL;appl:APPLICATION•
│        kernel ∈ secure_kernel ⇒ (construction (appl,kernel)) ∈ secure⌝);
```

The system echoes back the goal and awaits instructions on how to approach the proof of the goal.

```
ProofPower output
│(∗ ?⊢ ∗)  ⌜Z ∀ kernel : KERNEL; appl : APPLICATION
│              • kernel ∈ secure_kernel ⇒ construction (appl, kernel) ∈ secure⌝
```

Since the proof of the conjecture hinges entirely upon the definition of
secure_kernel we need to use the definition to expand the goal.

The procedure *z_get_spec* may be used to retrieve from the theory a theorem consisting of the conjunction of the predicate implicit in the declaration part of the specification and the explicit predicate. The combination is sufficient to justify rewriting the goal, but its form is not suitable for use by the rewriting facilities. This is because the explicit predicate is quantified, and its use depends upon establishing that the expression to be rewritten falls in the range of quantification. A similar HOL definition would suffice for rewriting, because quantification is permitted only over *types* and type checking alone establishes applicability of the definition. In Z, quantification is over sets (which may or may not be co-extensive with types), and type checking alone will not establish applicability of a quantified equation or equivalence; some proof is required.

In the case of axiomatic definitions of sets (i.e. properties) in this form it is possible to derive from the combination of the declaration and predicate parts of the specification an unconditional free variable form of the definition, which is more convenient for rewriting. A derived rule `iff_simp` is provided to undertake this derivation and its result is as follows. Since this equivalence is unconditional it may be used to expand out the goal using a general purpose rewriting tactic a follows:

SML
```
a (rewrite_tac[secure_kernel_sim]);
```

ProofPower output
```
(∗ ?⊢ ∗)  ⓩ∀ kernel : KERNEL; appl : APPLICATION
             • kernel ∈ KERNEL
                 ∧ (∀ appl : APPLICATION
                   • construction (appl, kernel) ∈ secure)
               ⇒ construction (appl, kernel) ∈ secure⌝
```

The goal has now become logically complex, and the proof may be progressed by repeating *z_strip_tac*, a tactic suitable for use on goals whose top level connective is a propositional connective or a universal quantifier.

SML
```
a (REPEAT strip_tac);
```

The resulting goal is in the form of a *conclusion* to be proven, together with a number of *assumptions* which are available for use in the proof. The assumptions are listed first, each enclosed in square brackets, followed by the conclusion to be established.

ProofPower output
```
(∗  3 ∗)  ⓩkernel ∈ KERNEL⌝
(∗  2 ∗)  ⓩappl ∈ APPLICATION⌝
(∗  1 ∗)  ⓩ∀ appl : APPLICATION • construction (appl, kernel) ∈ secure⌝

(∗ ?⊢ ∗)  ⓩconstruction (appl, kernel) ∈ secure⌝
```

Further progress now depends on choice of a suitable value for instantiation of the generalisation in the assumptions. This case is sufficiently simple that the instantiation will be found by *all_asm_fc_tac*.

SML
```
a (all_asm_fc_tac[]);
```

ProofPower output
```
Tactic produced 0 subgoals:
Current and main goal achieved
```

The theorem arising from proof of the conjecture may now be stored in the theory.

SML

```
save_pop_thm "architecture_secure";
```

ProofPower output

```
val it = ⊢ ∀ kernel : KERNEL; appl : APPLICATION
    • kernel ∈ secure_kernel ⇒ construction (appl, kernel) ∈ secure : THM
```

This very simple example has exhibited some of the basic machinery used for conducting proofs using the ICL Z proof tool.

It has also shown that conjectures about which subsystems of critical systems are themselves critical may be susceptible of formal proof. The fact that critical requirements were specified only on the kernel and not on the application indicates that in this architecture the application is not a critical component.

7 A Model of a Secure Kernel

7.1 The Model

We may now proceed to the definition of a function which we believe to be a *secure_kernel*. We call this an implementation for present purposes. In another context we might regard this as a functional specification or a design.

The specification is sufficiently explicit that it could be manually translated to a similar program in a functional programming language, though it is more plausible as a rather abstract model of a kernel which would be elaborated somewhat before implementation.

The kernel adopts two measures to ensure that the application does not violate the security policy (the critical requirement). Firstly it ensures that the application does not have access to information which the user is not cleared to see. This is modelled by the kernel supplying the application with a filtered copy of the classified data store from which highly classified data has been removed. Secondly it ensures that the application does not transfer information from highly classified data into data classified lower. This is modelled by the kernel filtering the classified data store returned from the application, discarding lowly classified data before using this to update the state of the system.

Z

```
│ kernel_implementation : KERNEL
├──────────────────────────────
│ ∀ clear:ℕ; inp:IN; state:STATE2; appl:APPLICATION •
│ kernel_implementation appl (clear, inp, state) =
│ ((state ⊕ ((0 .. (clear−1)) ⩤ (first (appl (inp, (0 .. clear) ◁ state)))),
│ second (appl (inp, (0 .. clear) ◁ state))))
```

7.2 The Correctness Proof

The fact that the *kernel_implementation* is a *secure_kernel* is not a trivial consequence of the definitions.

It is however a sufficiently straightforward consequence that full details of an interactive proof session establishing this result can be given. The level of complexity of the example was engineered with this in mind.

7.2.1 Mathematical Lemmas

A number of lemmas of a purely mathematical nature are required in this proof.

Some of these are results about set theory which are straightforwardly provable in the course of the main proof. A tactic is prepared here to deal with these proofs automatically.

The primary proof facilities available are of the following three kinds:

- stripping

  The tactic **z_strip_tac** embodies the basic techniques for dealing with propositional connectives, including skolemisation of quantifiers where appropriate.

  Repeating *z_strip_tac* progresses the proof through to the point at which further progress depends upon instantiation of generalisations in assumptions or on the provision of witnesses for existential conclusions of subgoals.

  In the case that the goal or subgoal is a propositional tautology this is automatically discharged by repeating *z_strip_tac*.

- resolution and forward chaining

  Identifying candidate values for instantiation of universal assumptions or witnesses for existential conclusions is typically undertaken in resolution based systems using unification. Resolution proofs are supported by ProofPower, but often "forward chaining" using the assumptions and pattern matching rather than unification provides a more effective method. These provide only for instantiation of universal assumptions (and stripping of the results) using pattern matching rather than unification. Despite these limitations they are effective in proving a useful class of predicate calculus results with minimal user intervention.

- rewriting

  General rewriting facilities are available in HOL, and these have been adapted for use with Z (the adaptations concern the "built-in" rewriting equations and the support of the more elaborate quantifiers in Z).

The system includes decision procedures for several useful problem domains, e.g., elementary set theory, typically packaged as proof contexts.

Two purely mathematical lemmas are required for the main proof, namely *le_dots_lemma1* and *le_dots_lemma2*. These results are obtained by a combination of rewriting and forward-chaining. We use the proof contexts for set-extensionality and numbers. These contain a variety of results which are applied automatically when rewriting or stripping the goal.

SML
```
set_pc "z_library_ext";
```

SML
```
set_goal ([], ⓩ∀ x, y : ℤ • x ≤ y ⇒ (0 .. x) ⊆ (0 .. y)⌝);
```

This result is proven by expanding the definition of .., stripping the result and then forward chaining using transitivity of ≤.

SML

```
a(rewrite_tac[z_get_spec ⓩ(_.._)⌝] THEN REPEAT strip_tac);
a(all_fc_tac[z_≤_trans_thm]);
```

SML

```
val le_dots_lemma1 = save_pop_thm "le_dots_lemma1";
```

SML

```
set_goal ([], ⓩ∀ x, y : ℤ • ¬ x ≤ y ⇒ (0 .. y) ⊆ (0 .. (x − 1))⌝);
```

The proof of this is more difficult than that of *le_dots_lemma1*. First expand the definition of ...

SML

```
a(rewrite_tac[z_get_spec ⓩ(_.._)⌝] THEN REPEAT strip_tac);
```

ProofPower output

```
...
(*  3 *)  ⓩy < x⌝
(*  2 *)  ⓩ0 ≤ x1⌝
(*  1 *)  ⓩx1 ≤ y⌝

(* ?⊢ *)  ⓩx1 ≤ x + ~ 1⌝
...
```

Now forward chain on the assumptions using *z_≤_less_trans_thm* to obtain $x1 < x$.

SML

```
a(all_fc_tac[z_≤_less_trans_thm]);
```

ProofPower output

```
...
(*  1 *)  ⓩx1 < x⌝

(* ?⊢ *)  ⓩx1 ≤ x + ~ 1⌝
...
```

Now it is necessary to expand the definition of < in the last assumption. *POP_ASM_T* takes out the last assumption and feeds it into the *THM_TACTIC* supplied to it. In this case we rewrite the assumption with the specification of < before passing it to *ante_tac*, which inserts in into the conclusion of the goal as the *ante*cedent of a new implication.

SML

```
a(POP_ASM_T (ante_tac o pure_once_rewrite_rule[z_get_specⓩ(_<_)⌝]));
```

ProofPower output

```
...
(* ?⊢ *)  ⓩx1 + 1 ≤ x ⇒ x1 ≤ x + ~ 1⌝
...
```

In the absence of support for linear arithmetic (which is available in ProofPower-HOL for natural numbers but not yet for integers in ProofPower-Z) this obvious result must be proven by transforming the conclusion of the goal until the various cancellation laws built into this proof context apply. First we move everything to the left hand side of the inequalities using *z_≤_≤_0_thm*.

SML
```
a(once_rewrite_tac[z_≤_≤_0_thm]);
```

ProofPower output
```
...
(∗ ?⊢ ∗) ⌜Z(x1 + 1) + ∼ x ≤ 0 ⇒ x1 + ∼ (x + ∼ 1) ≤ 0⌝
...
```

Now we use *z_plus_order_thm* to reorder the arithmetic expressions and *z_minus_thm* to provide some cancellation results which have been omitted from the proof context.

SML
```
a(rewrite_tac[z_∀_elim ⌜Z∼ x⌝ z_plus_order_thm, z_minus_thm]);
```

ProofPower output
```
...
Current and main goal achieved
...
```

SML
```
val le_dots_lemma2 = save_pop_thm "le_dots_lemma2";
```

In due course we hope that ProofPower will include support for linear arithmetic over integers in ProofPower-Z, similar to that currently supported for natural numbers in ProofPower-HOL.

SML
```
val ×_fc_thm = prove_rule []
        ⌜Z (∀ v:𝕌; w:𝕌; V:𝕌; W:𝕌 • v ∈ V ∧ w ∈ W ⇒ (v,w) ∈ (V × W))⌝;
```

7.2.2 Statement and Proof of the Proposition

Our objective is to prove that *kernel_implementation* is a *secure_kernel*.

Z
```
?⊢ kernel_implementation ∈ secure_kernel
```

SML
```
set_pc "z_sets_alg";
set_goal([],⌜Zkernel_implementation ∈ secure_kernel⌝);
```

ProofPower output
```
(∗ ?⊢ ∗) ⌜Zkernel_implementation ∈ secure_kernel⌝
```

First we expand the goal using definitions of *secure_kernel*, *secure*, *state_secure* and *out_secure*, and "strip" the resulting goal.

SML

```
val specs = map (z_defn_simp_rule o z_get_spec)
        [ⓩsecure_kernel⌝, ⓩsecure⌝, ⓩstate_secure⌝, ⓩout_secure⌝];
a (     rewrite_tac specs
        THEN REPEAT strip_tac);
```

ProofPower output

```
Tactic produced 6 subgoals:

(* *** Goal "6" *** *)
(*  6 *)  ⓩappl ∈ APPLICATION⌝
(*  5 *)  ⓩclear ∈ ℕ⌝
(*  4 *)  ⓩinp ∈ IN⌝
(*  3 *)  ⓩs ∈ STATE2⌝
(*  2 *)  ⓩs' ∈ STATE2⌝
(*  1 *)  ⓩ(0 .. clear) ◁ s = (0 .. clear) ◁ s'⌝

(* ?⊢ *)  ⓩ(construction (appl, kernel_implementation) (clear, inp, s)).2
             = (construction (appl, kernel_implementation) (clear, inp, s')).2⌝


...
(* *** Goal "4" *** *)
(*  7 *)  ⓩappl ∈ APPLICATION⌝
(*  6 *)  ⓩclass ∈ ℕ⌝
(*  5 *)  ⓩclear ∈ ℕ⌝
(*  4 *)  ⓩinp ∈ IN⌝
(*  3 *)  ⓩs ∈ STATE2⌝
(*  2 *)  ⓩs' ∈ STATE2⌝
(*  1 *)  ⓩ(0 .. class) ◁ s = (0 .. class) ◁ s'⌝

(* ?⊢ *)  ⓩ(0 .. class)
                ◁ (construction (appl, kernel_implementation)
                      (clear, inp, s)).1
             = (0 .. class)
                ◁ (construction (appl, kernel_implementation)
                      (clear, inp, s')).1⌝
```

ProofPower output

```
...
(* *** Goal "2" *** *)
(*  1 *)  ⓩappl ∈ APPLICATION⌝

(* ?⊢ *)  ⓩconstruction (appl, kernel_implementation) ∈ SYSTEM⌝
```

ProofPower output

```
...
(∗ ∗∗∗ Goal "1" ∗∗∗ ∗)
(∗ ?⊢ ∗)  ⌜Z kernel_implementation ∈ KERNEL⌝

The subgoal 2 duplicates goals labelled 3, 5
The subgoal 3 duplicates goals labelled 2, 5
```

The two subgoals listed first derive from the two properties (*out_secure* and *state_secure*) which a system must have in order to be secure.

The current subgoal (Goal “1”) consists of the predicate implicit in the declaration of *kernel_implementation*. We unpack the definition of *kernel_implementation* and *construction* for use in the proof.

SML

```
val [condec, conpred] = strip_∧_rule (z_get_spec ⌜Z construction⌝);
val [kidec, kipred] = strip_∧_rule (z_get_spec ⌜Z kernel_implementation⌝);
```

ProofPower output

```
val condec = ⊢ construction ∈ APPLICATION × KERNEL → SYSTEM : THM
val conpred =
   ⊢ ∀ appl : APPLICATION; kernel : KERNEL
        • construction (appl, kernel) = kernel appl : THM

:> val kidec = ⊢ kernel_implementation ∈ KERNEL : THM
val kipred =
   ⊢ ∀ clear : ℕ; inp : IN; state : STATE2; appl : APPLICATION
        • kernel_implementation appl (clear, inp, state)
          = (state
               ⊕ (0 .. clear − 1) ⩤ first (appl (inp, (0 .. clear) ◁ state)),
            second (appl (inp, (0 .. clear) ◁ state))) : THM
```

and then supply the relevant part to discharge the current subgoal.

SML

```
(∗ ∗∗∗ Goal "1" ∗∗∗ ∗)
a (strip_asm_tac kidec);
```

ProofPower output

```
Tactic produced 0 subgoals:
Current goal achieved, next goal is:

(∗ ∗∗∗ Goal "2" ∗∗∗ ∗)
(∗  1 ∗)  ⌜Z appl ∈ APPLICATION⌝

(∗ ?⊢ ∗)  ⌜Z construction (appl, kernel_implementation) ∈ SYSTEM⌝
...
```

This subgoal, is typical of the proof obligations arising in Z from the use of sets as if they were types. The decidable type-checking undertaken when the specification is entered leaves proof obligations of this kind.

SML
```
(* *** Goal "2" *** *)
a (asm_tac kidec THEN asm_tac condec);
a (LEMMA_T
        ⓩ(appl, kernel_implementation) ∈ (APPLICATION × KERNEL)⌝
        asm_tac
        THEN1 contr_tac);
```

ProofPower output
```
...
(*  2 *)  ⓩconstruction ∈ APPLICATION × KERNEL → SYSTEM⌝
(*  1 *)  ⓩ(appl, kernel_implementation) ∈ APPLICATION × KERNEL⌝

(* ?⊢ *)  ⓩconstruction (appl, kernel_implementation) ∈ SYSTEM⌝
...
```

SML
```
a (all_fc_tac [z_fun_∈_clauses]);
```

This discharges the current subgoal.

ProofPower output
```
Tactic produced 0 subgoals:
Current goal achieved, next goal is:
...

```

7.2.3 The *state_secure* Subgoal

The next subgoal corresponds to demonstrating that a system built using the kernel has the *state_secure* property.

ProofPower output

```
(* *** Goal "4" *** *)
(*  7 *)  ⌜appl ∈ APPLICATION⌝
...
(*  4 *)  ⌜inp ∈ IN⌝
...
(*  1 *)  ⌜(0 .. class ◁ s) = (0 .. class ◁ s')⌝

(* ?⊢ *)  ⌜(0 .. class
                ◁ (construction
                        (appl, kernel_implementation)
                      (clear, inp, s)).1)
            = (0 .. class
                ◁ (construction
                        (appl, kernel_implementation)
                      (clear, inp, s')).1)⌝
```

The subgoal is now progressed by expanding *construction* and *kernel_implementation*.

SML

```
(* *** Goal "4" *** *)
a (strip_asm_tac kidec);
a (ALL_FC_T asm_rewrite_tac [kipred, conpred]);
```

ProofPower output

```
...
(*  8 *)  ⌜appl ∈ APPLICATION⌝
...
(*  2 *)  ⌜(0 .. class) ◁ s = (0 .. class) ◁ s'⌝
(*  1 *)  ⌜kernel_implementation ∈ KERNEL⌝

(* ?⊢ *)  ⌜(0 .. class)
                ◁ (s ⊕ (0 .. clear − 1) ⩤ (appl (inp, (0 .. clear) ◁ s)).1)
            = (0 .. class)
                ◁ (s' ⊕ (0 .. clear − 1) ⩤ (appl (inp, (0 .. clear) ◁ s')).1)⌝
...
```

Now we have to think for a few moments to find a proof strategy.

If ⌜$clear \leq class$⌝ then ⌜$(0..clear) \subseteq (0..class)$⌝ and, given:

$$⌜(0..class) \lhd s = (0..class) \lhd s'⌝$$

we can conclude that:

$$⌜(0..clear) \lhd s = (0..clear) \lhd s'⌝$$

This fact may be used to rewrite the goal, changing the second occurence of s to s'. The resulting goal will be provable using:

$$⌜(0..class) \lhd s = (0..class) \lhd s'⌝$$

once more, with the theorem:

$$ⓩx \lhd z = x \lhd z' \Rightarrow x \lhd (z \oplus y) = x \lhd (z' \oplus y)⌝$$

If ⓩ$\neg clear \leq class$⌝ then ⓩ$0..class \subseteq 0..(clear - 1)$⌝, and the theorem:

$$ⓩ(A \subseteq B) \Rightarrow (A \lhd z) = (A \lhd z') \Rightarrow (A \lhd (z \oplus (B \lhd s))) = (A \lhd (z' \oplus (B \lhd s')))⌝$$

suffices to prove the subgoal.

A case split on the proposition ⓩ$clear \leq class$⌝ is therefore chosen.

This is the only point at which the proof is not routine. The proof tool does not necessarily help you to discover the proof plan; in general this can only be done by scrutinising the subgoal and coming to an understanding of why it is true. Describing the proof plan was more difficult than formulating it, the description was written after the proof had been completed, and completing the formal proof on the machine was not much more difficult than describing the informal proof on paper.

The structure of the proof is not unconnected with the the intuition behind the design of the kernel. This involves the use of two filtering operations on classified data stores. The case split used reflects the fact that for data at some classification just one of these filters is sufficient to ensure that this data item does not receive downgraded information. Which filter is needed depends upon whether the class is less than the clearance of the user. If it is, a filter prevents the data being updated, if it is not, the other filter ensures that no more highly classified data is used in the computation of the new value.

The following command initiates the case split:

SML
```
a (cases_tac ⓩclear ≤ class⌝);
```

This gives two subgoals differing only in their last assumptions:

ProofPower output
```
Tactic produced 2 subgoals:

(* *** Goal "4.2" *** *)
(*  9 *)  ⓩappl ∈ APPLICATION⌝
...
(*  3 *)  ⓩ(0 .. class) ◁ s = (0 .. class) ◁ s'⌝
(*  2 *)  ⓩkernel_implementation ∈ KERNEL⌝
(*  1 *)  ⓩ¬ clear ≤ class⌝

(* ?⊢ *)  ⓩ(0 .. class)
                ◁ (s ⊕ (0 .. clear − 1) ⩤ (appl (inp, (0 .. clear) ◁ s)).1)
              = (0 .. class)
                ◁ (s' ⊕ (0 .. clear − 1) ⩤ (appl (inp, (0 .. clear) ◁ s')).1)⌝
```

ProofPower output

```
(∗ ∗∗∗ Goal "4.1" ∗∗∗ ∗)
(∗  9 ∗)  ⓩappl ∈ APPLICATION⌝
...
(∗  3 ∗)  ⓩ(0 .. class) ◁ s = (0 .. class) ◁ s′⌝
(∗  2 ∗)  ⓩkernel_implementation ∈ KERNEL⌝
(∗  1 ∗)  ⓩclear ≤ class⌝

(∗ ?⊢ ∗)  ⓩ(0 .. class)
                ◁ (s ⊕ (0 .. clear − 1) ⩤ (appl (inp, (0 .. clear) ◁ s)).1)
            = (0 .. class)
                ◁ (s′ ⊕ (0 .. clear − 1) ⩤ (appl (inp, (0 .. clear) ◁ s′)).1)⌝
...
```

The first of the previously cited arithmetic lemmas, ($\vdash x \le y \Rightarrow 0..x \subseteq 0..y$) is now used:

SML

```
(∗ ∗∗∗ Goal "4.1" ∗∗∗ ∗)
a (fc_tac [rewrite_rule[z_get_specⓩℤ⌝]le_dots_lemma1]);
```

reducing the conjecture to a result which can be seen to follow from purely set-theoretic principles:

ProofPower output

```
...
(∗ 10 ∗)  ⓩappl ∈ APPLICATION⌝
...
(∗  4 ∗)  ⓩ(0 .. class) ◁ s = (0 .. class) ◁ s′⌝
(∗  3 ∗)  ⓩkernel_implementation ∈ KERNEL⌝
(∗  2 ∗)  ⓩclear ≤ class⌝
(∗  1 ∗)  ⓩ0 .. clear ⊆ 0 .. class⌝

(∗ ?⊢ ∗)  ⓩ(0 .. class)
                ◁ (s ⊕ (0 .. clear − 1) ⩤ (appl (inp, (0 .. clear) ◁ s)).1)
            = (0 .. class)
                ◁ (s′ ⊕ (0 .. clear − 1) ⩤ (appl (inp, (0 .. clear) ◁ s′)).1)⌝
```

The truth of this goal is most easily shown by first deriving and rewriting with the fact that ⓩ$(0..clear)\lhd s = (0..clear)\lhd s'$⌝. This fact is obtained by forward chaining using the following purely set-theoretic lemma which is proved automatically by the decision procedures in the proof contexts "set_ext_pcs".

SML

```
val **set_lemma_1** = pc_rule1 "z_rel_ext" prove_rule []
        ⓩ∀ A, B : 𝕌; x, x′ : 𝕌 •
        A ⊆ B ⇒ (B ◁ x) = (B ◁ x′) ⇒ (A ◁ x) = (A ◁ x′)⌝;
a (ALL_FC_T asm_rewrite_tac[set_lemma_1]);
```

ProofPower output

```
...
(* 10 *)  ⌜appl ∈ APPLICATION⌝
...
(*  4 *)  ⌜(0 .. class ◁ s) = (0 .. class ◁ s′)⌝
...
(*  3 *)  ⌜kernel_implementation ∈ KERNEL⌝
(*  2 *)  ⌜clear ≤ class⌝
(*  1 *)  ⌜0 .. clear ⊆ 0 .. class⌝

(* ?⊢ *)  ⌜
 (0 .. class) ◁ (s ⊕ (0 .. clear − 1) ⩤ (appl (inp, (0 .. clear) ◁ s′)).1)
 =
 (0 .. class) ◁ (s′ ⊕ (0 .. clear − 1) ⩤ (appl (inp, (0 .. clear) ◁ s′)).1)⌝
```

Similarly:

SML

```
val set_lemma_2 = pc_rule1 "z_rel_ext" prove_rule[]
        ⌜∀ A : 𝕌; x, x′, y : 𝕌 •
                A ◁ x = A ◁ x′ ⇒ A ◁ (x ⊕ y) = A ◁ (x′ ⊕ y)⌝;
a(ALL_FC_T asm_rewrite_tac[set_lemma_2]);
```

ProofPower output

```
Tactic produced 0 subgoals:
Current goal achieved, next goal is:
```

We now return to the second case in the proof of the *state_secure* property.

ProofPower output

```
(* *** Goal "4.2" *** *)
(*  9 *)  ⌜appl ∈ APPLICATION⌝
...
(*  3 *)  ⌜(0 .. class ◁ s) = (0 .. class ◁ s′)⌝
(*  2 *)  ⌜kernel_implementation ∈ KERNEL⌝
(*  1 *)  ⌜¬ (clear ≤ class)⌝

(* ?⊢ *)  ⌜(0 .. class)
                ◁ (s ⊕ (0 .. clear − 1) ⩤ (appl (inp, (0 .. clear) ◁ s)).1)
             = (0 .. class)
                ◁ (s′ ⊕ (0 .. clear − 1) ⩤ (appl (inp, (0 .. clear) ◁ s′)).1)⌝
```

The proof of this depends on the the second of the arithmetic lemmas:

$$(\vdash\neg(x{\leq}y){\Rightarrow}0..y{\subseteq}0..(x-1))$$

together with the non-trivial but automatically provable result from set theory shown below together with the assumptions.

SML

```
(* *** Goal "4.2" *** *)
val set_lemma_3 = pc_rule1 "z_rel_ext" prove_rule[]
        ⓩ∀ A,B:𝕌; x,x':𝕌; st,st':𝕌 •
        A ◁ x = A ◁ x' ⇒ (A ⊆ B)
        ⇒ A ◁ (x ⊕ (B ⩤ st)) = A ◁ (x' ⊕ (B ⩤ st'))⌝;
a (FC_T (MAP_EVERY ante_tac)
        [rewrite_rule[z_get_specⓩℤ⌝]le_dots_lemma2]
        THEN asm_ante_tac ⓩ(0 .. class) ◁ s = (0 .. class) ◁ s'⌝
        THEN rewrite_tac [set_lemma_3]);
```

This completes the subgoal relating to the *state_secure* property.

ProofPower output

```
Tactic produced 0 subgoals:
Current goal achieved, next goal is:
...
```

7.2.4 The *out_secure* Subgoal

The final subgoal corresponds to demonstrating that a system built using the kernel has the *out_secure* property.

ProofPower output

```
(* *** Goal "6" *** *)
(*  6 *)  ⓩappl ∈ APPLICATION⌝
(*  5 *)  ⓩclear ∈ ℕ⌝
(*  4 *)  ⓩinp ∈ IN⌝
(*  3 *)  ⓩs ∈ STATE⌝
(*  2 *)  ⓩs' ∈ STATE⌝
(*  1 *)  ⓩ(0 .. clear) ◁ s = (0 .. clear) ◁ s'⌝


(* ?⊢ *)  ⓩ(construction (appl, kernel_implementation) (clear, inp, s)).2
            = (construction (appl, kernel_implementation) (clear, inp, s')).2⌝
```

We are now left with the subgoal corresponding to the property *out_secure.* Once again we begin by rewriting with the specifications of *construction* and *kernel_implementation.*

SML

```
(* *** Goal "6" *** *)
a (MAP_EVERY asm_tac [condec, kidec] THEN
  ALL_FC_T asm_rewrite_tac [conpred, kipred]);
```

In this case this is sufficient to prove the subgoal and also completes the proof of the main goal.

ProofPower output

```
Tactic produced 0 subgoals:
Current and main goal achieved
```

SML

```
val kernel_secure_thm = save_thm ("kernel_secure1", pop_thm());
```

ProofPower output

```
val kernel_secure_thm = ⊢ kernel_implementation ∈ secure_kernel : THM
```

7.3 The System Correctness Proposition

Even though we have said nothing about the behaviour of the application, we have done enough formal modelling to establish that a system built from *kernel_implementation* and an application using *construction* will be secure.

If the application is loosely specified as:

Z

```
│      application : APPLICATION
├──────────────────────────────
│      true
```

and the system as:

Z

```
│      system : SYSTEM
├──────────────────────────────
│      system = construction(application, kernel_implementation)
```

The claim that this system is secure may then be expressed:

Z

```
│?⊢    system ∈ secure
```

The proof of this conjecture is trivial given the two previous results.

8 Concluding Remarks

8.1 Methods

We have given tiny illustrations of formal methods similar to those which we have used in our applications over the past few years, together with some explanation of why these methods have been thought desirable (which inevitably appear as criticisms of more widely accepted methods).

I hope that this will help to broaden the debate about what formal methods are good for, and to encourage the view that formal notations such as Z are a flexible resource which can and should be used to support a variety of different methods, varying according to the application domain and the particular concerns which the formal modelling is intended to address.

8.2 Tools

This paper has illustrated the use of the ProofPower Z proof tool developed by ICL to a point at which it is possible to reason about small systems with reasonable facility. Development of ProofPower, both for Z and for HOL continues, while the system is in use by ICL for applications requiring specification or proof in Z or HOL.

The challenge for non-critical systems is to bring automated reasoning about formal specifications to the point at which this can yield tangible *productivity* benefits. This demands not only developments in tools, but open minds and further innovation in methods and notations.

8.3 Acknowledgements

Both the methods and the tools are the result of teamwork, in a context in which publication of results and distribution of credit are subsidiary considerations. All the present and past members of the High Assurance Team at ICL Secure Systems have contributed to the work reported in this paper, as well as the many academics whose work we have exploited, particularly the groups at the Universities of Cambridge and Oxford who have been responsible for the development of HOL and Z.

The current members of the High Assurance Team are: Rob Arthan, Kevin Blackburn, Barry Homer, David King, Gill Prout, Roger Stokes and myself.

[1] R.D. Arthan. Formal Specification of a Proof Tool. In S.Prehn and W.J.Toetenel, editors, *VDM '91, Formal Software Development Methods, LNCS 551*, volume 551, pages 356–370. Springer-Verlag, 1991.

[2] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[3] Michael J.C. Gordon. HOL:A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis.* Kluwer, 1987.

[4] Michael J.C. Gordon and Tom F. Melham, editors. *Introduction to HOL.* Cambridge University Press, 1993.

[5] Michael J.C. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF. Lecture Notes in Computer Science. Vol. 78.* Springer-Verlag, 1979.

[6] Jeremy Jacob. On The Derivation of Secure Components. *Proc. 1989 IEEE Symposium on Security and Privacy*, pages 242–247, 1989.

[7] R.B. Jones. Methods and Tools for the Verification of Critical Properties. In R.Shaw, editor, *5th Refinement Workshop*, Workshops in Computing, pages 88–118. Springer-Verlag/BCS-FACS, 1992.

[8] R.Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[9] J.M. Spivey. *Understanding Z.* Cambridge University Press, 1988.

[10] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1989.

Index of Formal Names