Z Styles for Security Properties and Modern User Interfaces

Anthony Hall

Praxis Critical Systems, 20 Manvers Street, Bath BA1 1PX
anthony.hall@praxis-cs.co.uk

Abstract. This paper describes two new styles for using Z. The first style, based on earlier work for the UK Government, is suitable for the specification of security properties in the shape of a formal security policy model. The second, an extension of the Established Strategy, is useful for specifying systems with modern graphical user interfaces and also for showing satisfaction of security properties. The work is based on a successful industrial project.

1 Introduction

1.1 Background

One of the best-known formal notations, Z, consists of a small collection of set theoretic and logical symbols together with a structuring tool, the schema. The semantics of Z are mathematical, not computational. This makes it a powerful, flexible and extensible notation. However, since there is no built-in correspondence between the meaning of a Z specification and any computational model, it means that use of Z to specify computing systems is a matter of convention. One dominant convention has grown up - the so-called Established Strategy. However, other styles are also possible and some of these other styles are more useful for expressing particular kinds of property. In particular, a convention for using Z to define security properties was developed at UK Government's Computer and Electronic Security Group (CESG) [4]. Barden, Stepney and Cooper have published a practical guide [5], which gives good accounts of both the Established Strategy and the CESG approach.

The CESG method for specifying secure systems is as follows:

1. Write a model which expresses the security policy desired (the Formal Security Policy Model, FSPM).
2. Write a specification of the system to be built (the Formal Top Level Specification, FTLS).
3. Prove a theorem about the correspondence between the FTLS and the FSPM.

In the FSPM, the system is specified as a transition relation between inputs plus starting state and final state plus output. The security properties are expressed as constraints on this system. These constraints define that subset of all systems which are secure systems.

The FTLS, on the other hand, is first expressed in the Established Strategy as a state plus a collection of operations, each operation defined by a schema. The method then transforms this, by a relatively mechanical process, to a transition relation similar to the FSPM. To carry out step 3, the FSPM is instantiated with the types of the FTLS and an interpretation function is written which interprets the behaviour of the FTLS as a system in terms of the FSPM. The proof is constructed in two steps: first each operation is considered separately to generate *Operation Lemmas* and then properties of the total transition relation are proved from the operation lemmas.

1.2 An Example Project

During 1998 Praxis Critical Systems developed a Certification Authority (CA) for MULTOS. The development was carried out according to the principles of Information Technology Security Evaluation Criteria (ITSEC) level E6, although the system was not formally evaluated. ITSEC E6 requires a formal security policy and a formal design of the system. There is a description of the overall development process and its results in [6] and a more detailed account of the role of formal methods in the project in [7]. As part of this development we wrote an FSPM and an FTLS, following the basic principles of the CESG method. We did not try to prove correspondence between them. Given the size of the system (well over 100 operations) this would have been a major task. However, we did write the specifications in such a way that this proof could be attempted in future.

We based our FSPM on the CESG method. However the security policy for the CA required particular operations to be supplied. It also depended on more complex properties of data than the mandatory security classifications which were the concern of [4]. We therefore adapted the CESG method in two respects: we modelled the system as a set of operations rather than a transition relation, and we had a more elaborate model of the properties of data.

The CA has a modern user interface using windows, command buttons and selection from lists. The Established Strategy has a simple model of inputs and outputs, which does not represent such an interface faithfully. Many of the security properties of the CA, however, were constraints on this interface (for example that no secrets were displayed). It was therefore important, if we were to establish these properties, that they should be represented in the FTLS. We therefore extended the conventions of the Established Strategy to represent important aspects of the user interface.

2 A Formal Security Policy Model

2.1 What We Had to Formalise

The user requirements included an informal security policy, which identified assets, threats and countermeasures. We formalised a subset of the whole policy.

Of the 45 items in the informal policy, 28 were technical as opposed to physical or procedural. Of these, 23 items related to the system viewed as a black box, whereas 5 were concerned with internal design and implementation details. We formalised these 23 items, turning them into 27 formal clauses which fell into three classes:

- Two of the clauses constrained the overall state of the system. Each of these became a state invariant in the formal model.
- Eight clauses required the CA to perform some function (for example, authentication). To formalise this we need to say that there must exist an operation with a particular property.
- Seventeen clauses were constraints applicable to every operation in the system (for example, that they were only to be performed by authorised users). For these we need to say that during any operation execution where the clause is applicable, the property must hold.

2.2 Overall Approach

We followed the overall approach of the CESG method by defining:

1. A general model of a system.
   We model any system as a state plus a set of operations; this is slightly different from the CESG approach and the differences are discussed in more detail in section 2.3.
2. A model of a CA system
   This is a specialisation of the general model of a system with some mapping between system state and real-world concepts such as users, sensitive data and so on. This idea is taken from the CESG approach where the SYSTEM schema includes not just the transition relation but also application-specific concepts such as clearance. However, we implement the idea rather differently, using representation functions to relate system data to real-world constructs, as described in section 2.4.
3. A definition of a secure CA system.
   Each formalisable clause in the security policy is turned into a predicate that constrains the CA system in some way. The overall definition of a secure CA system is one where all the constraints are satisfied. Section 2.5 gives an example of how we formalised each type of property.

2.3 A General Model of A System

The approach here differs from that in [4] in that we do not define the transition function of the system explicitly. Instead, we define a system in terms of a state and a *set of operations*. The difference is purely formal: given a constraint over our set of operations, we could turn it into a constraint over a transition relation using the method described in [4]. We have not done this because none of the items in the security policy need constraints over sequences of operation applications. Therefore the constraints over the transition relation would reduce directly

into constraints over single operation applications - the *Operation Lemmas* of [4].

The system state is simply a collection of data.

$$[DATA]$$
$$State == \mathbb{F}\ DATA$$

We assume a set *OPERATION* which can be used to refer to individual operations. Each operation may succeed or fail: if it fails, it returns one or more errors.

$$[OPERATION, ERROR]$$

Each operation execution has some input. Each operation execution may produce output to be displayed and output to be transmitted outside the CA on some medium such as CDROM. These two kinds of output are distinguished. The item *display* refers to anything which may appear on the screen as a result of an operation. This may include the echo of input values, as well as any results or error messages from the operation. The item *transmitted* refers to any output which is to be transmitted as a result of an operation.

OperationExecution

$$state, state' : State$$
$$operation : OPERATION$$
$$input : \mathbb{F}\ DATA$$
$$display : \mathbb{F}\ DATA$$
$$transmitted : \mathbb{F}\ DATA$$
$$errors : \mathbb{F}\ ERROR$$

We define a system as a set of possible states, a set of possible initial states, a set of possible operations and a collection of possible operation executions.

System

$$states : \mathbb{P}\ State$$
$$initialStates : \mathbb{P}\ State$$
$$operations : \mathbb{F}\ OPERATION$$
$$opExecutions : \mathbb{P}\ OperationExecution$$

$$initialStates \subseteq states$$
$$\forall\ oe : opExecutions \bullet oe.state \in states$$
$$states = \{OperationExecution \mid \theta OperationExecution \in opExecutions \bullet$$
$$\quad state \mapsto state'\}^*(\!|initialStates|\!)$$
$$\{o : opExecutions \bullet o.operation\} = operations$$

The variable *operations* is the identities of the operations that the system supports. Note that these operations must actually be possible in the system. We do not allow systems to contain operations which can never be executed from any reachable state. This is of course a rather weak liveness condition, but it corresponds to the notion that certain operations must exist in the system.

The possible states are just the initial states plus those reachable from the initial states by sequences of the possible operation executions.

It would be a mechanical process to transform *opExecutions*, the set of all possible executions, into the transition relation used to characterise a system in [4].

2.4 A Model of A CA System

This section gives a general definition of a CA system. It describes every possible CA system, both secure and insecure. It has to be rich enough for us to write predicates which distinguish secure from insecure systems. It therefore has to contain all the concepts which are used in the security policy.

The CESG approach defines the system in terms of generic types: specialisation to a particular kind of system is done by instantiating these generic types. Our problem did not lend itself to this approach, since the security properties of the system depend on a number of different kinds of data representing, for example, secret information, users, audit trails and so on.

Instead of having a generic type *DATA* therefore, we used a given set *DATA* and defined relevant properties of data in one of two ways:

1. Some of the security policy can be expressed in terms of very general properties of data such as whether it is secret. Such properties can be expressed simply as subsets of *DATA*.
2. To define other parts of the policy, we need to understand what some parts of the data in the system actually represent. For example, we need to know that some data represent role holders and their properties. In that case, what we do is construct a separate model (similar to the FTLS) of the relevant parts of the real world, and construct functions which map *DATA* to this real world model.

General properties of data For the purpose of this example, we consider two aspects of data: whether it is or is not secret and whether or not it needs some kind of confidentiality, non-repudiation or integrity protection. An important property is that the set of secret data is fixed: there are no operations which can change the classification of data.

$$secret : \mathbb{P}\ DATA$$

For the purpose of this example, we simplify the need for protection to just two cases: data are either sensitive or insensitive. We also omit details of how mechanisms are combined.

$$sensitive, insensitive : \mathbb{P}\ DATA$$
$$\langle sensitive, insensitive \rangle \ \mathsf{partition}\ DATA$$

Protection consists of applying some mechanism to a piece of data. If a data item has been protected, it is no longer sensitive. Protection must of course be

reversible - it must be possible to recover the original data, if the protection mechanism is known.

$[MECHANISM]$

$$
\begin{array}{|l}
protection, recovery : MECHANISM \longrightarrow DATA \nrightarrow DATA \\
\hline
\forall\, m : MECHANISM;\ d : DATA \mid d \in \mathrm{dom}(protection\ m) \bullet \\
\quad (protection\ m)\, d \in insensitive \land \\
\quad (recovery\ m)((protection\ m)\, d) = d
\end{array}
$$

A data item cd is a (possibly protected) copy of a data item d if it is either d itself or the result of applying some protection to d.

$$
\begin{array}{|l}
\_ \ \mathsf{copyOf}\ \_ : DATA \leftrightarrow DATA \\
\hline
\forall\, cd, d : DATA \bullet cd\ \mathsf{copyOf}\ d \Leftrightarrow \\
\quad (d = cd \lor (\exists\, m : MECHANISM \bullet cd = (protection\ m)\, d))
\end{array}
$$

Mapping Data to the Real World Some parts of the policy can not be expressed simply as properties of data. Instead, we have to consider how the data represents security-relevant things in the real world. We do this as follows:

1. Identify the relevant real world entities and their relationships. For simplicity in translating to items of data, relationships are all represented as simple binary relations (which may of course be functions).
2. Define *representation functions* which map *DATA* into these real world entities. An item of data represents either an instance of an entity, in which case the state includes a set of such entities, or a pair of entities, in which case the state includes a set of such pairs: that is, a relation between two entity types. For example an item of *roleHolderData* represents a single role holder; an item of *roleHolderNameData* represents a (role holder, name) pair.

As an example, part of the policy deals with roles and role holders. The real-world structure is described in the schema *Access*

$[ROLEHOLDER, ROLE, TEXT]$

$$
\begin{array}{|l}
\textit{Access} \\
\hline
roleHolders : \mathbb{F}\ ROLEHOLDER \\
roles : ROLEHOLDER \leftrightarrow ROLE \\
\hline
\mathrm{dom}\ roles = roleHolders \\
\hline
\end{array}
$$

This schema contains a well-formedness constraint that all role holders do in fact have roles.

To represent this, we map role holders and roles to items of data. The variable *roleHolderData* represents the role holders and *roleHolderRolesData* represents the roles held by role holders.

$$roleHolderData, roleHolderRolesData : \mathbb{P}\ DATA$$

$$\mathsf{disjoint}\ \langle roleHolderData, roleHolderRolesData \rangle$$

$$rRoleHolder : roleHolderData \rightarrowtail\!\!\!\!\!\!\rightarrow ROLEHOLDER$$
$$rRoleHolderRoles : roleHolderRolesData \rightarrowtail\!\!\!\!\!\!\rightarrow ROLEHOLDER \times ROLE$$

A state represents a value of the schema *Access* when the relevant parts of the state are exactly those produced by representing the values of the variables in *Access*.

$$rAccess : State \pfun Access$$

$$\begin{aligned} rAccess = \{ & s : State; a : Access \mid \\ & a.roleHolders = rRoleHolder(\!|s|\!) \land \\ & a.roles = rRoleHolderRoles(\!|s|\!) \bullet s \mapsto a \} \end{aligned}$$

A CA system is one in which the states do indeed correspond to the structure of *Access*.

CASystem

$$System$$

$$\forall s : states \bullet s \in \mathrm{dom}\ rAccess$$

2.5 The security policy

The security policy is a conjunction of predicates which constrain the system. A system which satisfies all the predicates in the policy is secure according to that policy.

A State Invariant: One role only The informal policy states that it will be impossible for an individual to assume more than one role at any one time. This is formalised by saying that each role holder can only have one role.

We formalise this in two stages: first we formalise the real-world constraint:

OneRoleReal

$$Access$$

$$roles \in ROLEHOLDER \pfun ROLE$$

Then we state the property that the CA System itself must represent a real world situation which respects the constraint. That is, every state in the system must represent a real world which respects the constraint.

OneRoleOnly

$$CASystem$$

$$\forall s : states \bullet rAccess\ s \in OneRoleReal$$

The Existence of an Operation: Backup Availability The informal policy states that it shall be possible to continue operations on another site in the event of system failure. This requires two operations: save and restore. (Other features of the CA preclude any other solution to the informal requirement.) In this (simplified) definition *save*, provided it is successful, generates a (possibly protected) copy of all data in the state. This copy may contain additional administrative information, but it must contain at least the state.

SavePossible

$$
\begin{array}{l}
System \\
\hline
\exists\, save : operations \bullet (\forall\, o : opExecutions \mid \\
\quad o.operation = save \land o.errors = \varnothing \bullet \\
\quad\quad (\forall\, d : o.state \bullet (\exists\, cd : o.transmitted \bullet cd \ \mathsf{copyOf}\ d)))
\end{array}
$$

A Property of All Operations: Data Transmission The informal policy requires that the system will ensure that any data that are transmitted over a communications channel are afforded the security of fit-for-purpose confidentiality, integrity and non-repudiation mechanisms. The formal policy states that any transmitted data is insensitive: that is it is an original item which does not need protection or it has been protected.

ProtectTransmittedData

$$
\begin{array}{l}
System \\
\hline
\forall\, o : opExecutions \bullet \\
\quad o.transmitted \subseteq insensitive
\end{array}
$$

The Secure CA System The secure CA system is defined as a CA system where all the clauses of the policy are respected:

SecureCASystem

$$
\begin{array}{l}
CASystem \\
\hline
OneRoleOnly \\
SavePossible \\
ProtectTransmittedData
\end{array}
$$

2.6 Relation to the FTLS

The formal top level specification must conform to the formal security policy model. To show conformance, the following steps are necessary:

1. For every data structure used in the FTLS, identify the data in the FSPM that represent it.

2. Where the data in the FSPM represent some real world concept such as ROLEHOLDER, define the representation function in terms of FTLS structures. In some cases the FTLS structure will be identical to the real world structure used to define the FSPM, but it is not necessary that the correspondence is the identity.
3. Using these correspondences, provide a mapping between possible states in the FTLS and the states in the FSPM, and show that all the states allowed by the FTLS conform with the predicates in the FSPM which constrain the system state.
4. Translate all the operation definitions in the FTLS into operations in the FSPM which represent them. An FSPM operation represents an FTLS operation if the FSPM data forming the inputs, outputs and state changes in the FSPM represent the inputs, outputs and state changes of the FTLS operation. All operations in the FTLS can be represented in the FSPM by translating their data in this way.
   Then show that all such FSPM operations conform to the predicates which constrain all operations.
5. Where the FSPM calls for the existence of a particular operation, demonstrate that there is a corresponding operation in the FTLS. Translate the FTLS operation into an FSPM operation which represents it, and demonstrate that the definition conforms with the definition of the required operation.

3 The FTLS

The structure of the FTLS is driven by two factors:

1. The operation definitions must be structured so that it is easy to translate them into the corresponding FSPM structures.
   This means that we must clearly state, for each operation, what is input, what is displayed and what is transmitted out of the CA system. The display must include any error messages; there is a requirement that all errors are reported, so we must allow for more than one error per operation.
2. The CA has a graphical user interface. This means that operations are chosen from a limited set available on the screen, and the operations available at any time depend on the state of the system. Furthermore once an operation has been selected, there is a dialogue for the user to give the inputs to the operation. Inputs may be selected from a limited set of possibilities (for example in a list box) or they may be typed in by the user.
   This means that we have to represent the fact that operations may not be available, that they are long-lived and that some inputs may not be available to the user.

We use three conventions to meet these requirements.

1. We have separate schemas for the inputs, displayed items and transmitted items for each operation.
   This allows us to match the operation specifications to the FSPM.

2. We have separate schemas to define what inputs to an operation are available, what inputs are correct and what inputs are invalid.
   This models the fact that some inputs are selected from a limited choice of possibilities.
3. We specify the execution of an operation in two phases. There is a generic specification *StartOperation* which models the selection of an operation by the user, and leaves the system in a state where that operation is active. There is then a specific definition for each operation describing its behaviour. This captures the fact that only certain operations are available at any one time, and the fact that operations are long-lived rather than atomic. It also allows tracing to rules in the FSPM which govern when certain operations may take place.

As an illustration, we use the operation to add a role holder. In this simplified specification we assume that the state consists simply of two maps, from role holder ids (which are text) to their passwords and roles.

The only other independent component of the state is the current operation. This may or may not be present, and to represent this we use a convention (which seems to have been invented simultaneously by several authors) of representing optional items as sets containing either zero or one member.

$$\text{optional}\, X == \{x : \mathbb{F}\, X \mid \#x \leq 1\}$$
$$nil[X] == \varnothing[X]$$
$$the[X] == \{\, x : X \bullet \{x\} \mapsto x \,\}$$

CAState

$$roleHolderPassword : TEXT \pfun TEXT$$
$$roleHolderRole : TEXT \pfun ROLE$$
$$known : \mathbb{F}\, TEXT$$
$$currentOperation : \text{optional}\, OPERATION$$

$$known = \text{dom}\, roleHolderPassword = \text{dom}\, roleHolderRole$$

There is a general schema *OperationFrame* which defines the state change for all operations. Since this schema represents completion of the operation, there is no operation current in the final state.

OperationFrame

$$\Delta CAState$$

$$currentOperation' = nil$$

3.1 Inputs and Outputs

We need to state exactly what the inputs for each operation are, what is displayed on the screen and what is transmitted to outside systems. Furthermore

we need to be able to extract this information systematically from the specification. Therefore, for each operation *xxx*, we specify up to three schemas: *xxxIn*, *xxxDisp* and *xxxXmit*.

xxxIn contains the inputs specific to the operation. These are decorated with "?".

xxxDisp contains *everything* significant that is displayed on the screen. The only items that are not considered "significant" in this context are prompts and similar items which are fixed and not dependent on the data in the system. This is so we can check that the system does not display anything which contradicts the security policy. This includes any listboxes which are put up for the user to choose from, and any echoes of user input. Where it contains echoes of user input, it repeats the name of the input variable. If the item displayed is not an echo of input, it is decorated with "!" in the usual way. *xxxDisp* always contains the declaration $error! : \mathbb{F}\ ERROR$, which is used to report all errors to the user. If the operation succeeds, *error*! is always empty. We sometimes include part of the state in *xxxDisp* so as to describe constraints on the diplay. The state components are not, of course displayed: the only items that are actually displayed are those which appear in *xxxDisp* and are decorated with "?" or "!".

xxxXmit contains everything that is output on other media, such as floppies, DAT tapes and CD ROM. This is decorated with "!" as usual.

The state change, inputs and outputs are collected together in a frame schema *xxxFrame*. This schema also defines parts of the state which are unchanged by the operation.

For example, here is a simplified specification of the operation to add a new role holder to the system.

The inputs are a role holder id which is typed in as text, a role which is selected from a given list of roles and a password which is typed in as text.

RegisterRoleHolderIn

$roleHolderId? : TEXT$
$role? : ROLE$
$password? : TEXT$

The display includes an echo of the role holder id and the selected role, but not of the password. It also includes the set of roles available. In this simplified example we omit the predicate defining what roles are actually displayed.

RegisterRoleHolderDisp

$roleHolderId? : TEXT$
$role? : ROLE$

$role! : \mathbb{F}\ ROLE$
$error! : \mathbb{F}\ ERROR$

In this particular case there is nothing transmitted outside the CA so we omit the schema *RegisterRoleHolderXmit*

RegisterRoleHolderFrame

OperationFrame
RegisterRoleHolderIn
RegisterRoleHolderDisp

---

the currentOperation = *registerRoleHolder*

3.2 Availability and Validity

An operation will only have certain inputs available. For example, it might be physically impossible to provide invalid input values if input is done by selection from a menu. The conditions where the operation can be attempted are described in a schema *xxxAvailable*

In some cases, it is possible to try the operation with particular parameters, but it won't work. The rules for correct input are defined in a schema *xxxValid*. The behaviour of the operation in this case is defined in the schema *xxxOK*. Conversely, the schema *xxxError* defines the effect of invoking the operation with invalid inputs. The total operation is then $xxxOK \lor xxxError$.

For example, in *RegisterRoleHolder* the user selects a role from the list of displayed roles.

RegisterRoleHolderAvailable

RegisterRoleHolderFrame

$role? \in role!$

The user can type any role holder id and password they like. However, the id they type must not already exist.

RegisterRoleHolderValid

RegisterRoleHolderAvailable

$roleHolderId? \notin known$

Successful operation depends on the input being valid. The *RegisterRoleHolderOK* schema describes the effect in that case.

RegisterRoleHolderOK

RegisterRoleHolderValid

$roleHolderPassword' = roleHolderPassword \oplus \{roleHolderId? \mapsto password?\}$
$roleHolderRole' = roleHolderRole \oplus \{roleHolderId? \mapsto role?\}$

$error! = \varnothing$

The error schema describes what happens in case each of the validity conditions is not met. It allows more than one error to be reported, and in fact is loose in that other, implementation-defined, errors are also possible.

RegisterRoleHolderError

$$RegisterRoleHolderAvailable$$
$$\Xi CAState$$

$$roleHolderId? \in known \Leftrightarrow theRoleHolderIdHasBeenUsed \in error!$$

The total definition of *RegisterRoleHolder* is:

$$RegisterRoleHolder \mathrel{\widehat=} RegisterRoleHolderOK \lor RegisterRoleHolderError$$

3.3 The Lifecycle of an Operation

The previous sections have defined how an operation behaves once it has been invoked. The action of invoking an operation is treated separately. There is an operation *StartOperation*. This describes how a user may attempt to start any operation which is available at the time. It succeeds provided that the rules for that operation are satisfied.

The only input to *StartOperation* is the operation id.

StartOperationIn

$$operation? : OPERATION$$

At certain times the user is offered a list of operations they can invoke. The user can then select an operation to be run from the listed operations.

We don't define here all the rules for what operations are displayed, but certainly none are displayed if there is already a current operation. We include the *currentOperation* part of the state to allow us to represent that fact, but since it is not decorated, *currentOperation* is not actually displayed.

StartOperationDisp

$$StartOperationIn$$
$$operations! : \mathbb{F}\ OPERATION$$
$$currentOperation : \text{optional}\ OPERATION$$

$$currentOperation \neq nil \Rightarrow operations! = \varnothing$$

StartOperationFrame

$$OperationFrame$$
$$StartOperationDisp$$

An operation is only available to be started if it is one of those on the screen.

StartOperationAvailable

$$StartOperationFrame$$

$$operation? \in operations!$$

An operation can only be started when certain conditions are met. We omit the actual definition of *StartOperationValid*, but it captures rules such as only allowing certain operations if certain roles are present.

StartOperationValid

$$StartOperationAvailable$$

If the operation is valid, it becomes the current operation.

StartOperationOK

$$StartOperationValid$$

$$the\ currentOperation' = operation?$$

4 Summary

This paper describes two new styles for using Z.

The first, which is relevant to the specialist security community, is a variant on the approach developed for Computer and Electronic Security Group. It allows for a wider range of security properties, by supporting a richer chracterisation of data and by making operation types explicit. It should also remove the need for one proof step, although to justify that claim we would have to complete a proof using our approach.

The second style is more generally applicable to the specification of modern software systems. It is an extension to the Established Strategy which provides a richer model of inputs and outputs and a more faithful representation of a typical modern user interface.

These styles have been used on a real commercial project and were successful in allowing us to represent important properties of the system we were building. The security policy forced us to consider security properties in the Formal Top Level Specification, and the FTLS structure supported the specification of the user interface and the subsequent development of the system. The outcome of the development was a system which has proven robust and accurate in commercial use.

One important next step to validate this work would be to carry out proofs within this framework. We have worked out the strategy which would be needed to prove conformance of the FTLS with the Formal Security Policy Model, but not attempted any of the proofs.

We did typecheck all our Z with *f*UZZ. (This paper has also been typechecked, although we omitted some constant definitions to avoid cluttering the exposition.)

Acknowledgements

We thank John Beric, Head of Security at Mondex International, for permission to publish this work.

References

1. J.M Spivey, *The Z Notation: A Reference Manual*, Prentice Hall, Second Edition, 1992
2. B. Potter, J. Sinclair and D. Till, *An Introduction to Formal Specification and Z*, Prentice Hall, 1991
3. *Multos GKC System User Requirements*, Issue 1-9, 4 September 1997
4. *CESG computer security manual "F": A formal development method for high assurance systems*, Issue 1.1, July 1995
5. R. Barden, S. Stepney and D. Cooper, *Z In Practice*, Prentice Hall, 1994
6. A. Hall and R. Chapman, *Correctness by Construction: Developing a Commercial Secure System*, IEEE Software, Jan/Feb 2002, pp18 - 25.
7. A. Hall, *Correctness by Construction: Integrating Formality into a Commercial Development Process*, Proceedings of International Symposium of Formal Methods Europe, LNCS 2391, Springer, pp224 - 233.