UNLIMITED

(2)

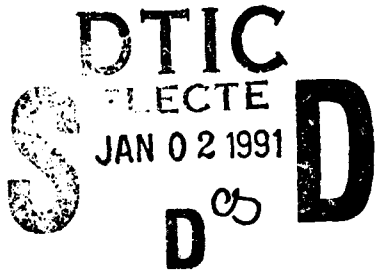
Report No. 90011

Report No. 90011

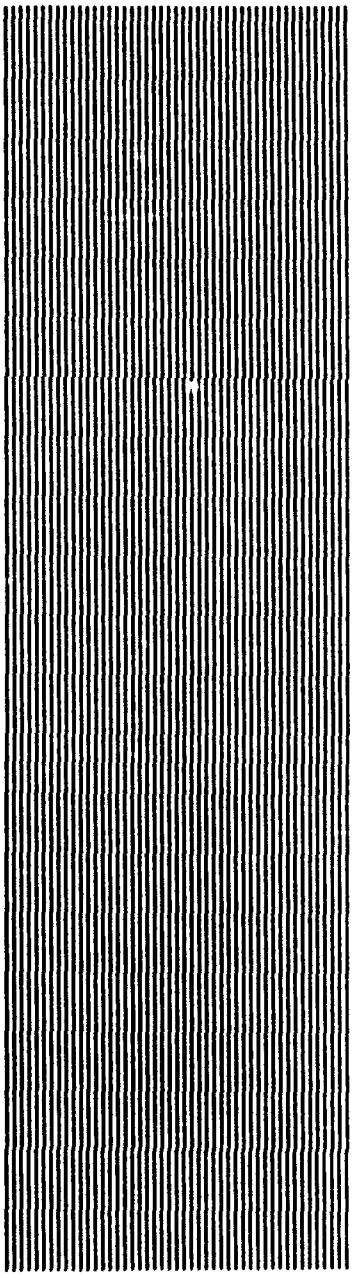
ROYAL SIGNALS AND RADAR ESTABLISHMENT, MALVERN

AN EXAMPLE SECURE SYSTEM SPECIFIED USING THE TERRY-WISEMAN APPROACH

Author: C L Harrold



PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.



July 1990

CONDITIONS OF RELEASE

0083040 BR-115326

********** DRIC U

********** DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report 90011

Title: An Example Secure System Specified Using the Terry-Wiseman Approach

Author: C L Harrold

Date: July 1990

Abstract

This report presents the specification of operations for a secure document handling system (SERCUS). The specification uses the Terry-Wiseman Security Policy Model and therefore acts as an example of the modelling approach. The specification uses the mathematical notation Z, and consequently also acts as an example of the use of Z in specifying secure systems. However, it must be noted that an appreciation of SERCUS, the model and modelling approach can usefully be gained even if the formal specifications are not read. The Terry-Wiseman Model and its interpretation are given as an Annex to this report.

Contents

1. Introduction

This paper formally presents a representative set of operations from the SERCUS application in terms of the Terry-Wiseman Security Policy Model [AnnexB,Terry&Wiseman89,Terry89,Harrold90a].

SERCUS is one aspect of the SMITE programme [Bottomley&Wiseman88,Harrold89]. SMITE is addressing the problems of the provision of high functionality, multi-level secure systems, which require a high degree of assurance that the desired security properties have been achieved. SERCUS is a research implementation of a multi-level secure workstation running a classified document handling system, whose purpose has been to show the soundness and feasibility of the SMITE Approach.

The requirements of SERCUS were originally specified, using the formal notation Z, in [Harrold88]. This specification was produced before the modelling work was completed, and is consequently a mixture of functionality and security requirements. The overall security requirement is that classified information cannot be discovered by users with insufficient clearances, ie the confidentiality of information is ensured. The consequences of this requirement on the functionality were included into the original specification in a rather intuitive manner, and thus it is not obvious that it did uphold the security requirement. The Terry-Wiseman Model defines what it means for a system to uphold confidentiality in a coherent framework, and can capture the implications of the functionality requirements upon the overall goal of confidentiality. Therefore, the SERCUS application has been modelled using the Terry-Wiseman Model. This report provides the specification of a representative sample of the SERCUS operations in terms of this model.

SERCUS is essentially an electronic registry system which controls the creation of, and access to, classified documents and mail messages. In the usual way, the users are assigned clearances which limit their ability to observe and modify the information in the system. In addition to their clearance, the users have a designated role to play. The possible roles are security officer and ordinary user, although there were also registry clerks in the original, longer, specification. Certain operations may only be performed by users with the appropriate role. For example, only security officers may create new legal users or review journalled information and, in the original specification, only registry clerks could create files or add documents to files. Although the model does allow systems to be specified where individuals can have more than one role, this is not required in the SERCUS application, and each user is assigned a single fixed role.

In addition to the roles that the human users of SERCUS can be assigned, there are three additional roles that certain software can possess. The first of these represents the software acting on behalf of the system owners, for example the login software which ensures that only people with authorisation from the owners can use SERCUS. Secondly, there is a role representing the trusted software which maintains high water mark classifications, and lastly, a role representing software that is trusted as a result of an evaluation process.

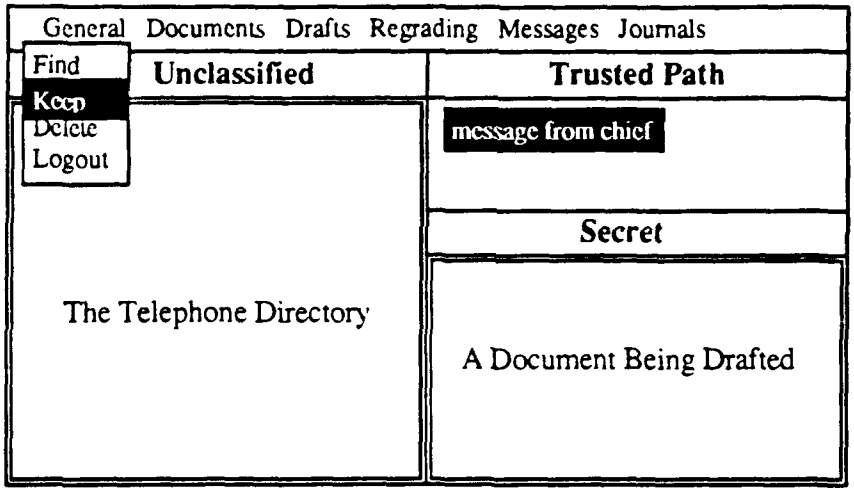
All users have a personal cupboard in which they may store certain kinds of objects, such as the documents they are drafting. Whilst in the cupboard these objects may be referred to by an unclassified name. SERCUS also maintains an unclassified list of all the finished classified documents, called the classified document register (CDR). Users may observe the CDR and ask to read any of the documents contained in it. An additional requirement of documents is that their classification may be altered. However, to ensure that they are not downgraded inappropriately, a security officer and a user must agree the new classification.

In SERCUS a journal is maintained for each document. This is used to record the interesting events that have happened to the document, for example which users have accessed its contents and the users who agreed to any alteration to its classification. In order to hold the users accountable for their actions, a journal of the security relevant actions of each of them is also maintained. Some examples of the types of event that this journal records are: when the users log on and off SERCUS; any documents they were prevented from seeing because their clearance was insufficient; and any users they send mail messages to.

When logged on to SERCUS the user is presented with a display consisting of a number of windows [Wisemanetal88]. All the window software is completely trustworthy (a Trusted Path†), but may be used to invoke untrusted software, such as a commercial word processing package. While untrusted software is active in a window the classification of information is prominently displayed. SERCUS monitors the movement of objects and text between these windows and uses a high water mark mechanism [Woodward87] to correctly maintain the classification levels. The Trusted Path has been formally specified in [Wood88], also using the Z notation.

All windows in SERCUS are non-overlapping, or tiled. In other words, windows cannot be wholly or partly contained within or hidden by other windows. This is because it is very important that untrusted software cannot spoof the user in any way, for example by mimicking the Trusted Path interface and tricking the user into revealing a password. SERCUS uses non-overlapping windows because they are a simple way to prevent such spoofing. For example consider the following diagram, Figure 1, of a typical SERCUS display. Both the draft document and telephone directory are being displayed and edited using separate invocations of untrusted software, and the high water mark of each is displayed by the Trusted Path software above. The untrusted software is only given access to the rectangle of screen below this classification and therefore should it try to fool the user by displaying the words 'Trusted Path', it can easily be seen that this is within an untrusted area. Similarily, menus and dialogue boxes always operlap a portion of the screen accessible only to the Trusted Path software.

Figure 1: A typical display in SERCUS



The Terry-Wiseman Model has arisen out of the SMITE research programme. It provides a framework for defining the security requirements of a system in terms of its functionality, and can therefore provide a formal coherent framework for discussions between the customers, designers and developers of a secure system, helping to ensure that they all interpret the requirements in the same way. The formal model and its interpretation are given as Annex B, which is essentially section 3 from [Harrold90a].

The main area that Terry-Wiseman addresses is that of maintaining the confidentiality of information, in other words ensuring that unauthorised people cannot discover classified information. In addition, Terry-Wiseman provides for the integrity of the controls used to enforce confidentiality. It is an important feature of the model that both the information and its controls are described in such a way that the interactions between them can be considered. The model also permits alterations to security controls, and enforces n-man rules (separation of duty) to ensure that all such changes are appropriate. One of the major contributions of the model is that it identifies the particular aspect of trust that software must be shown to uphold, when implementing operations that could affect the confidentiality of information.

† A Trusted Path is a validated link between the human user and a system's trusted software which mutually authenticates both parties.

The specification language used throughout this report is Z [Spivey88,King*etal*87]. An overview of the notation is given in Annex A. This report is in fact a fully typechecked Z document [Sennett87,Randell90], in the form of separately type checked modules of Z. Each module is defined using a 'keeps' statement which lists the items to be made available to other modules. The modules 'used' by a piece of Z are included at the start and appear in the text with a box drawn around them, for example,

Some_Z :Module

For ease of reading, this is generally followed by a list of the Z definitions of interest that are imported from the module. This report therefore also acts as an example of the use of Z, and the advantages of a modular system for specifications.

Following this introduction, section 2 outlines the model and the approach taken for the specification exercise. The objects required in SERCUS are formally defined in section 3. Section 4 provides the operation specifications for a representative sample (of ten) of the operations in SERCUS. These are the operations to login and logout; create a window; create and edit draft documents; create, open and regrade classified documents; keep objects in the cupboard under a name and find named objects from the cupboard. Section 5 discusses some of the implications of the operation specifications, the operations that have been omitted and a summary of the modelling exercise.

It must be noted that although sections 3 and 4 do contain the formal specification, they also contain descriptive English text and diagrams. However, for those readers unsure of Z, it must be noted that an appreciation of SERCUS, the model and modelling approach can usefully be gained even if the actual formal specifications are not read.

2. Modelling Approach

2.1 Overview of the Model

The Terry-Wiseman Security Model considers that the state of any computer system can be captured by the relationships between sets of *entities* and *attributes*. Entities represent the containers of information, and are alterable objects in that their contents may change. Entities can represent containers at any level of abstraction, for example individual register locations, documents or directories. Attributes are immutable objects and represent the information itself, for example the integer 42 stored in a register, the textual contents of a document, the file names in a directory or a classification such as 'confidential'. Writing to a register is modelled as replacing it's contents attribute with another and not by altering the meaning of its initial contents. Similarly if the classification of an object is changed, this is modelled by replacing its classification attribute with another.

The relationships between entities and attributes may only be altered by state transitions. Furthermore, state transitions are modelled as having been initiated by a set of entities, called the *requestors*. This is a set, rather than a single entity, in order to capture the notions of separation of duty. Two further sets of entities are important when considering the information flows arising from a state transition. These are the *observed* and *modified* entities. The observed entities are those whose contents were in any way involved in the outcome of the transition. The modified entities are those whose view of the state, ie their attributes, were altered by the transition.

As mentioned earlier the model is concerned with ensuring that the confidentiality of information is upheld. In other words controlling the flows of information that take place whenever a state transition occurs. Thus, some of the attributes of entities are identified as control attributes, for example, the classification, degree of trust or role. Relations and functions are defined on the state to discover these security relevant attributes of an entity. These attributes are used to decide whether or not a particular transition is allowed. For example, a transition to copy attributes from a secret container to an unclassified container would not uphold the confidentiality requirement and would not be allowed to take place.

The protection mechanism in Terry-Wiseman is entities with control attributes, and therefore there are potential signalling channels through the existence and controls of all entities. It is always possible to probe a protection mechanism and discover 'something' about the controls. Consequently, the Terry-Wiseman Model makes the simplifying worst case assumption that the existence and control attributes of all entities are freely visible. Thus there is a further constraint on transitions to ensure that the existence and controls of entities are not being used to signal information, ie are not classified.

There are five axioms controlling the various types of information flows arising from transitions, and these are fully described, both formally and in English, in Annex B. Thus the security requirements are captured by modelling secure transitions, ie defining the ways in which entities may gain or lose attributes and the conditions under which they may be created or destroyed.

2.2 Using the Model

Essentially the modelling approach is to specify the objects required by the system in terms of entities and attributes, as is done for SERCUS in section 3. This can be considered to be typing and structuring the abstract definition of a system given in the general model. Each of the required operations of the system may then be specified in terms of the changes it makes to the relationships between these entities and attributes, as a state transition. In other words, an important step in the modelling approach is to specify exactly what the desired functionality of the system should be. This is done for the subset of SERCUS in section 4. The definition of the desired functionality is then combined with the security axioms of the model, resulting in the specification of a secure transition.

However, not all the required functionality results in a useful specification. Where the specified functionality is contradictory to the definition of security, the precondition of the secure transition becomes false, and the transition can never take place. Therefore, in some cases the required operation has to be specified as a sequence of individually secure transitions, separating out the flows of information or downgrading it. Where the functionality requirement is in fact 'insecure', the

requirement can either be altered, or the 'insecurity' considered to be not a risk, and documented accordingly in the system development.

It is not a particularly difficult task to see where the desired functionality is contradictory to the definition of security. Each of the axioms of the model can be considered in turn, and its implications examined. It is not always necessary to have a complete formal specification of the desired operation to see the security implications in terms of the model. The simple rule is that the system is secure, in terms of confidentiality, if the classification of modified entites dominates the observed information, ie there is no downward flow. Should this not be the case, and the operation still be desired, Terry-Wiseman requires that a justification for the downward flow be given in the form of separation of duty considerations. Note that this does not always require the physical cooperation of n-people every time the transition is invoked but can be 'delegated' to trusted software [Harrold90b].

3. SERCUS Functionality

This section describes the objects required in SERCUS in terms of entities and attributes, and gives a brief explanation of their desired properties. Essentially this section is typing and structuring the abstract sets of entities and attributes that were introduced in the policy model (Annex B). The English descriptions generally precede the formal Z specification, and, where words are in italics they refer to the formal definition.

First the definitions of entities and attributes, state transitions, etc, and the definition of a total ordering (not in the type checker's library) are included into this section by the the appropriate Z modules (as discussed in section 1).

policy_model :Module

Defines
E, A, REF, ROLE, CONFLICT, conflict_roles, CLASS, >=, GLB LUB, ID, TRANSITION, ↑, entities

defns_not_in_library :Module

Defines
total_order

3.1. Documents and the Classified Document Register

As in the paper world, all documents in SERCUS are centrally recorded on a Classified Document Register (CDR), and documents may be uniquely identified by their CDR number. The requirement is that the CDR numbers and the classification of documents are not themselves classified. Thus, an entity to model the CDR and a set of attributes to represent the CDR numbers are identified. The ordering on CDR numbers is given by *newer*.

$CDR : E$

$CDR\_NUM : \mathbb{P}\, A$

$\_\ newer\ \_ : total\_order_{[\ CDR\_NUM\ ]}$

A set of attributes to represent the entries in the CDR is identified. The particular reference and CDR number can be discovered from an *ENTRY* attribute by the function *entry*. This is an injective function (one to one) as the CDR numbers and references are unique to each *ENTRY* attribute. The requirement is that the *CDR* only contains references to documents and not to any other types of object, and hence the function is not surjective (onto).

$ENTRY : \mathbb{P}\, A$

$entry : ENTRY \rightarrowtail (CDR\_NUM \times REF)$

It is important to note that the structure of information in the CDR is captured by modelling *ENTRY* attributes and a function to reveal the structure, rather than the CDR simply containing CDR number and reference attributes. This is because the latter approach would not yield the structure and association between the CDR numbers and particular references, and, more importantly any alteration in the relationship between CDR numbers and documents would not result in a change in the functionality relationships between entities and attributes. In other words, there would be no change in the *Other* relation from the *STATE* schema and the change would not be captured by the model's definition of a transition. This technique of defining functions on attributes to reveal structure is used fairly extensively, and is further discussed and illustrated in section 5.

A set of entities is identified as representing documents. The requirements of documents are that they have attributes representing their CDR number, textual contents and a journal in which to record the interesting or security relevant events that have happened to them. Textual contents are modelled using *TEXT* attributes, see section 3.4, and journals are discussed in section 3.5. Documents are also required to be regradable. However, to ensure that the new classification is appropriate, regrades must be approved by both a security officer and a user.

$DOCUMENT : \mathbb{P}\, E$

An additional requirement of documents is that they have unalterable contents. This is analogous to not allowing Tippex (or any other brand of typing correction fluid) nor writing in the margins in the paper world. Consequently, no transitions will be specified which change the *TEXT* attribute of a document. Another consequence of this requirement is that any references in the text of documents are only to unalterable objects. Thus documents may only contain references to other documents.

3.2. Draft Documents

Documents will be created from draft documents. Thus, drafts are basically documents that have not yet been given their final classification. A set of entities is identified to represent these drafts.

$DRAFT : \mathbb{P}\, E$

Since the users of SERCUS may type in information up to their clearance, draft entities will be classified (ie their contents protected) at the clearance of the particular user. However, SERCUS requires that users be able to create documents lower than their clearance, and therefore maintains a high water mark for each draft which monitors the classification of information that it may contain. The draft may then be turned into a new document with an appropriate classification between the high water mark and the clearance of the user. See section 3.10 for the properties and maintenance of high water marks.

3.3. Windows

The users of SERCUS perform operations in the windows of a display, ie windows model the software running in the computer which is the proxy of the human user. Thus windows are modelled as active entities, the classification and roles of which will generally be those of the user in question. Windows will also be labelled with the id of the user they belong to, and one of their functionality attributes will be details about the user (see section 3.9). The *TRUST* attributes of the window will depend on the trustworthiness of the particular software in control. For example, a Trusted Path window is *faithful* and *dont_signal* (as defined in Annex B) since it is known that its actions originate from the human. An off-the-shelf word processing package would not be given any *TRUST* attributes as no guarantees can be made about what it actually does. Whenever such untrusted software is in control of a window the classification of the information it has accessed will be monitored using a high water mark (see section 3.10). A set of entities is identified as representing windows.

$WINDOW : \mathbb{P}\, E$

3.4. Contents

Some objects in SERCUS, eg documents and windows, contain textual contents which is a mixture of both references to further objects and characters. The set of attributes which represent this text is identified.

$TEXT : \mathbb{P}\, A$

The ability to discover the set of references contained in some text is required, and *refs_of* is identified for this purpose. This is a function since a *TEXT* attribute may contain only one set of references. It is a total function as all possible *TEXT* attributes are representing some set of references, possibly the empty set. The function is not injective (one to one) as each set of references may be viewed as contained in many texts, for example by altering and changing orders of characters or simply altering the order of the references as they appear to users. The function is not surjective (onto) as not all possible references may be in text.

$$refs\_of : TEXT \rightarrow \mathbb{P}\, REF$$

Note that further functions could be defined to discover more about the representation of a *TEXT* attribute, for example the textual characters or the order of references. However, at this high

level of specification, simply knowing which references are available in the text and being able to notice whenever any changes are made is sufficient.

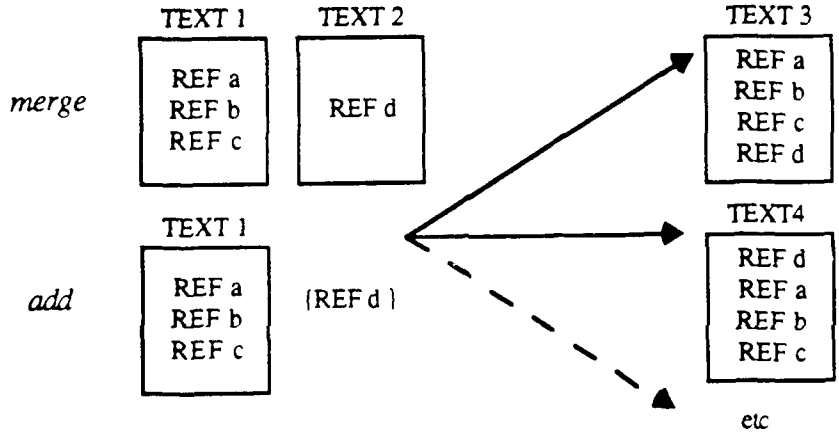
A particular *TEXT* attribute to represent blank text is identified. There are no references associated with blank text.

$$blank\_text : TEXT$$

$$refs\_of(\ blank\_text\ ) = \{\}$$

In many operations information is added to the text of an object, ie the original *TEXT* attribute of an entity is replaced by one representing more. The relations *merge* and *add* are defined for this purpose. Figure 2 illustrates the properties specified about these relations. Thus when texts are merged there are a number of possible resulting texts, each of which contain all the references from the unmerged texts but which differ in their orderings, etc. Similarily there are a number of ways references can be added to text.

Figure 2: Merging TEXTs and Adding References to TEXT



merge is a relation because there are many different ways of merging the same texts. However, at this level of specification the only interesting property of a TEXT attribute representing the result is that the set of references is the union of the individual sets.

$$merge : \mathbb{P}\ TEXT \leftrightarrow TEXT$$

$$\forall\ t : \mathbb{P}\ TEXT;\ r : TEXT \mid (t,r) \in merge \bullet refs\_of(\ r\ ) = \bigcup refs\_of(\!|\ t\ |\!)$$

Similarly, *add* is the relation which identifies the various texts that can result from adding a set of references to some text. Again, the only interesting property at this level of specification is that the references in the resulting text are the union of the original with the added set.

$$add : (\ TEXT \times \mathbb{P}\ REF\ ) \leftrightarrow TEXT$$

$$\forall\ t, r : TEXT;\ set : \mathbb{P}\ REF \mid ((t,set),r) \in add \bullet refs\_of(\ r\ ) = refs\_of(\ t\ ) \cup set$$

As with the contents of the classified document register, the text of objects is another case where modelling information by simply associating attributes for the references and characters would be insecure because alterations in structure and ordering would not be captured by any changes in the functionality relation.

3.5. Journals

In SERCUS some objects maintain a journal of the interesting or security relevant events that have happened to them. The security relevant operations that each user performs are also recorded. Thus, the set of all possible journal entities and the set of all possible event attributes are introduced. The representation of events indicates the particular user, type of event, time, etc, but is not of interest at this level of the specification. However, it is important that operations at these lower levels of abstraction preserve the 'attributeness', ie immutability, of the higher level abstraction. For example, altering the type of the event at the lower level of abstraction must result in a different *EVENT* attribute at the higher level, so that the change is captured by the transition axioms and the security implications can be considered.

$JOURNAL : \mathbb{P}\, E$
$EVENT : \mathbb{P}\, A$

Journals must be classified system high, or *top* (see section 3.8) as they may be written to whenever operations are performed, or sometimes when only requested, by both trusted and untrusted software, and with a wide range of clearances. This potential signalling channel was highlighted by the modelling exercise. Journals are not active entities requiring role or trust attributes.

Since journals are highly classified the requirement to audit them, ie observe their contents, needs to be thoroughly considered. The simplest solution is for certain users, for instance an auditor or security officer, to log in with a clearance of top. However, this would mean that this user is cleared to see all information in the system, which is not generally required. It is not appropriate to downgrade the journal itself, as this then allows untrusted lowly classified entities to observe its contents and potentially receive any encoded information. The approach taken by SERCUS is for the observation of a journal to be modelled by a sequence of transitions which create an entity classified *top*, copy in the relevant information from the journal, downgrade this new entity to a suitable classification, for example 'Auditor' or 'Security Officer Eyes Only', and then copy it into a Trusted Path window for the human user to review.

3.6. Cupboards

The users of SERCUS each have a personal cupboard in which they may store objects, such as documents and draft documents. Whilst in the cupboard these objects may be referred to by a name. Thus, cupboards are containers of information and are modelled as entities. The set of all possible cupboard entities is identified. Cupboards contain name and reference associations, and a set of attributes is identified as representing all possible items in the cupboard.

$CUPBOARD : \mathbb{P}\, E$
$ITEM : \mathbb{P}\, A$

The set of all possible names for items in a cupboard is introduced as a basic type, and a function defined, *item*, to associate *ITEM* attributes to the name-reference pair they are representing. This is a total function as every *ITEM* attribute is representing a single name-reference pair. It is injective (one to one) as each pair is unique to an *ITEM* attribute. Note that not all possible references may be kept in cupboards, and hence the function is not surjective (onto).

$[\, NAME \,]$

$item : ITEM \rightarrowtail (NAME \times REF)$

The additional requirement of a cupboard is that the names and references in it are not classified themselves, although the referenced objects may be. Thus, cupboards will be unclassified entities. There is no requirement to regrade a cupboard, and nor are cupboards active entities requiring roles or trust attributes.

3.7. Roles and Conflicts

There are only two possible roles of the human users of SERCUS, namely security officers and ordinary users. *ROLE* attributes are identified to represent these. Note that in SERCUS there is no requirement for a human user to play both roles.

sso, user : ROLE

As stated earlier, regrading a document requires that both a user and a security officer agree that the new classification is appropriate. Thus, it is useful to identify a *CONFLICT* attribute to represent these roles.

sso_user : CONFLICT

---

conflict_roles(sso_user) = { sso ↦ 1, user ↦ 1 }

The role of software acting on behalf of the system owners, for example the login software, is identified. Alterations to certain controls are justified by agreement with the system owners (see section 4.2), and therefore two further attributes are identified. These specify that agreement between the owner's proxy and a user, and the owner's proxy and a security officer are required.

system_owners_proxy : ROLE

owner_user, owner_sso : CONFLICT

---

conflict_roles(owner_user) = { system_owners_proxy ↦ 1, user ↦ 1 }
conflict_roles(owner_sso) = { system_owners_proxy ↦ 1, sso ↦ 1 }

Where there is no requirement to alter any of the control attributes of an entity, the conflict is specified as requiring the agreement of at least one of all possible roles (and not just the roles that are identified for SERCUS). To guarantee this lack of agreement to a change, one of these roles would be represented by some trusted software which never agreed to anything.

no_alteration : CONFLICT

---

dom (conflict_roles(no_alteration)) = ROLE

Some changes to the classification of an entity will be performed on the basis of its high water mark classification. For example a document may be given a final classification lower than the clearance of the user but no lower than the high water mark of the draft it was created from. This downgrade is justified by the separation of duty between the particular user and the trusted software which maintains high water marks. Thus, a *ROLE* attribute to identify the software managing the high water marks and two *CONFLICT* attributes are identified. These specify that agreement between the high water mark manager and a user, and the high water mark manager and a security officer are required.

hwm_manager : ROLE

hwm_user, hwm_sso : CONFLICT

---

conflict_roles(hwm_user) = { hwm_manager ↦ 1, user ↦ 1 }
conflict_roles(hwm_sso) = { hwm_manager ↦ 1, sso ↦ 1 }

In some cases changes to classifications are necessary to provide the desired functionality, but cannot be justified using high water marks. This is modelled by requiring that the evaluator agree with the system designers that the justifications are acceptable. Thus the evaluator role and an appropriate *CONFLICT* attribute are identified.

evaluator : ROLE

$$the\_evaluator : CONFLICT$$

$$conflict\_roles(\ the\_evaluator\ ) = \{\ evaluator \mapsto 1\ \}$$

It is useful to note that the evaluator role indicates an assurance issue and the system owners role an operational issue. Thus, for example, the evaluator provides assurance to the system owners that the login code works correctly, but is not aware of the users authorised by the system owners to login using this code. The use of the evaluator role is further discussed in section 5.

3.8. Useful Classifications

The following classifications are useful to define; *bottom* is the classification that is dominated by all other classifications; *unclassified* is the lowest classification possible for information in SERCUS, ie is dominated by all classifications except bottom; and *top* is the highest classification. For completeness, the classification of the greatest lower bound, *GLB*, and least upper bound, *LUB*, when applied to empty sets are defined.

$$bottom, unclassified, top : CLASS$$

$$\forall\ a : CLASS \bullet a >= bottom$$
$$top >= a$$
$$unclassified >= bottom$$
$$\forall\ c : CLASS \mid c \neq bottom \bullet c >= unclassified$$
$$GLB\ \{\} = top$$
$$LUB\ \{\} = bottom$$

3.9. User Details

A subset of all possible attributes is identified as representing information about the legal users of SERCUS. In the usual way passwords are required to authenticate the users when they request to login to SERCUS. Thus the set of all possible passwords are introduced.

$$USER : \mathbb{P}\ A$$
$$[\ PASSWORD\ ]$$

The following schema, *USERDATA*, contains the information that the *USER* attributes are representing. This is the user's identity (which must be unique to them), their password, clearance and role, a reference to a journal of their security relevant or interesting activities and the reference to their cupboard entity. Ensuring that each user has a different cupboard, unique identity, etc, is discussed in section 5.

USERDATA

$$uid : ID$$
$$password : PASSWORD$$
$$clearance : CLASS$$
$$role : ROLE$$
$$journal, cupboard : REF$$

A function, *user_data*, is defined to discover this representation of a *USER* attribute. This is a total function as each *USER* has a single representation and there are attributes for all possible users. It is injective (one to one) so that each *USERDATA* is only representing a single user.

$$user\_data : USER \rightarrowtail USERDATA$$

The systems owners cannot be present to agree, or otherwise, every request to use their system. Therefore they set up a system of separation of duty using trusted software acting on their behalf

and passwords to authenticate the users. This trusted software is modelled by the *LOGIN* entity which has as its functionality, attributes for all the legal users of SERCUS. The integrity of this user data is vital to the secure operation of the system, and can be modelled by the absence of undesirable functionality transitions. In other words, there will be no transitions, other than logging in and creating new users, which observe or modify this data. This integrity requirement is discussed further in section 5.

$LOGIN : E$

3.10. High Water Marks and the Trusted Functionality

As mentioned earlier, SERCUS maintains high water marks to monitor the activities of untrusted software and subsequently to justify some downgrades. Thus, a set of attributes are identified as modelling high water marks and a function defined, *class_of*, to supply the particular classification associated with the high water mark. This is a total function as all high water marks have a classification. Also, several high water marks may have the same classification, and not all possible classifications need necessarily be associated with high water marks.

$HWM : \mathbb{P}\, A$

$class\_of : HWM \rightarrow CLASS$

Note that high water marks have been identified as a separate set from the classification attributes. This is to enable there to be other classification attributes, for example the required classification of a new document, amongst the functionality attributes of entities without the dangers of confusing them in the design or implementation. In addition merging entities together may be modelled.

The use of high water marks to justify downgrades can itself only be justified when they are correctly maintained. The general requirement is that whenever an entity with a high water mark is modified, the high water mark is raised to the level of the high water marks, or classifications as appropriate, of all observed entities. This is a property required of all transitions, and, in the same way as the security requirements of the policy model, this is specified as an axiom defining constraints on state transitions, called *Trusted_Func*. In order to simplify the specification this axiom splits the functionality relation, ie the *Other* part of the *STATE* schema, into two, namely high water marks, *Hwm*, and the remainder, *Func*.

It is useful to remember that the policy model defines a *TRANSITION* to be a request, $r?$, consisting of those entities which requested the transition and those which were observed, together with the *STATE* of the system before and after the transition, s and s' respectively. Also note that the definition of *modified* includes any newly created entities, but excludes those destroyed by a transition.

It is not required that all entities in the state have high water marks, hence *Hwm* is partial. However, whenever any entity with a high water mark is *modified* by a transition, the high water mark ought to be raised to reflect the classification (or high water mark if they have one) of all *observed* entities. If the high water mark is not raised in this way, then the evaluator must have been involved in the transition and agreed that this downgrade was appropriate. High water marks cannot be removed from entities.

The high water mark of an entity may only be altered if some of its other functionality attributes were also altered. The *Trusted_Func* axiom ensures that whenever an operation specifies alterations to the functionality of some entities, associated high water marks will be raised correctly. It is not appropriate to explicitly state in all operation specifications that the high water marks of all other entities are not changed. Any changes to these high water marks would have obeyed all the axioms, however the requirement is that the high water marks only exist to monitor modifications to the remainder of the functionality.

Trusted_Func

TRANSITION

$Hwm, Hwm' : E \pfun HWM$

$Func, Func' : E \rel A$

$Hwm = s.Other \rhd HWM$

$Hwm' = s'.Other \rhd HWM$

$Func = s.Other \rhd HWM$

$Func' = s'.Other \rhd HWM$

$\forall m : modified \bullet$

$m \in dom\ Hwm' \Rightarrow$

$(\neg (Hwm'(m) >= LUB\ class\_or\_hwm(\!| r?.observed \cup \{m\} |\!)) \Rightarrow evaluator \in s.Role(\!| r?.requestors |\!))$

$m \in dom\ Hwm \Rightarrow$

$m \in dom\ Hwm'$

$Hwm(m) \neq Hwm'(m) \Leftrightarrow (Func(\!|\{m\}|\!) \neq Func'(\!|\{m\}|\!))$

where

$modified == dom\ (s'.Other \upharpoonright s.Other) \cap entities\ s'$

$class\_or\_hwm == s.Class \oplus (Hwm \fatsemi class\_of)$

3.11. Composing Transitions

Some operation requirements cannot be modelled as a single transition as this would violate the *no_flows_down* security axiom. In these cases the requirement has to be modelled as a sequence of transitions separating out the flows of information or downgrading information. A desire for functional integrity requires that information be passed between these transitions to ensure that later transitions in the sequence act upon the results of previous ones. The level of abstraction of this specification is not concerned with the details of which attributes are part of this flow and how they are used. This is an implementation issue which, to an extent, depends upon the chosen representation of abstract attributes, and is further discussed in section 5. Therefore, this specification simply identifies a set of attributes to represent all possible *FLOW* information that an implementation could require.

$FLOW : \mathbb{P}\ A$

As this level of specification is not concerned with how an implementation uses flow information, it is sufficient to simply identify in a transition specification those entities which may be given flow information but leave the choice of attribute non-deterministic, except to the extent that it is governed by the security axioms. In other words, the specification still identifies all the *modified* entities and requires that their attributes are only derived from the nominated *observed* entites and the unclassified control information.

Thus, the *Functional_Integrity* axiom identifies the *FLOW* attributes from the general functionality by a function, *Flow*, and uses this to identify the set of entities whose flow information was altered by the transition, *changed_flows*. Since *Flow* is a function, entities are constrained to have at most one *FLOW* attribute, and as it is a partial function not all entities in the state need have *FLOW* attributes. The remainder of the application functionality is identified by the *Appl* relation. Thus, this axiom has further structured the functionality of the state that was defined by the *Trusted_Func* axiom above.

Functional_Integrity

Trusted_Func
changed_flows : $\mathbb{P}\ E$
Appl, Appl' : $E \leftrightarrow A$

---

changed_flows = *dom* (*Flow* $\uparrow$ *Flow'*) $\cap$ *entities s'*

where

Flow, Flow' : $E \nrightarrow FLOW$

Flow = *Func* $\triangleright$ *FLOW*
Flow' = *Func'* $\triangleright$ *FLOW*

Appl = *Func* $\triangleright$ *FLOW*
Appl' = *Func'* $\triangleright$ *FLOW*

3.12. Some Properties

In order to be able to distinguish between kinds of entity, the various types defined above must be disjoint. Attribute types need not be disjoint, as it is permissable for the same attribute to represent different things depending upon location. For example the integer 1 may represent 'Top Secret' when in the *Class* function and 'January' when in *Other*.

disjoint ⟨ *CUPBOARD, JOURNAL, WINDOW, DOCUMENT, DRAFT,* {*CDR*}, {*LOGIN*} ⟩

The following defines section 3 to be a module of Z, and exports the definitions for use in the operation specifications of section 4.

Functionality keeps CUPBOARD, JOURNAL, WINDOW, DOCUMENT, DRAFT, LOGIN, CDR_NUM, CDR, ITEM, TEXT, EVENT, ENTRY, HWM, USER, FLOW, NAME, item, PASSWORD, USERDATA, refs_of, merge, add, blank_text, newer, entry, class_of, user_data, sso, user, system_owners_proxy, owner_user, owner_sso, no_alteration, hwm_manager, bottom, evaluator, sso_user, hwm_user, hwm_sso, top, the_evaluator, unclassified, Trusted_Func, Functional_Integrity

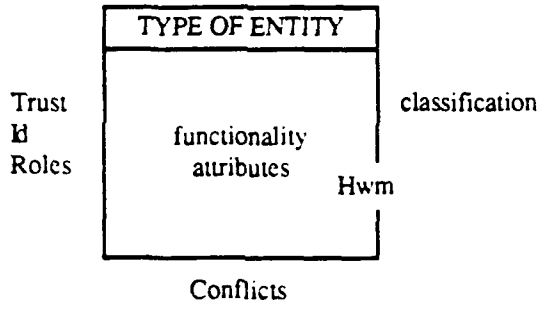
4. Operations

This section provides the formal specification of ten of the more interesting operations in SERCUS. Section 4.1 first defines some useful schemas that are used by the other specifications.

Each of the operation specifications begins with an overview of the functionality requirements, for example whether the event is journalled, or what types of entity are created and where their attributes come from. Where the requirement is slightly more complex, a diagram is given to illustrate the entities and attributes involved in the operation and the relationships between them.

The following diagram, Figure 3, is a pictorial representation of an entity and its attributes. The entity is drawn as a box containing functionality attributes of interest. The classification of the entity will be written to the right of the box with its vertical position indicating the relative classification. The position of any high water mark inside the box also indicates the relative classification. Any trust, id or role attributes will be written to the left of the box, and the conflicts for the separation of duty controls written below. Information will be omitted from the diagram when it is not relevant to the discussions.

Figure 3: Pictorial Representation of an Entity



Note that any arrows between entities represent references and that dashed arrows represent the references to entities created by the operation under consideration.

Following the overview of the operation, the functionality requirements are fully specified by a Z schema (or several schemas). This uses the naming convention, operation*_functionality*. This schema is not concerned with any of the security requirements, except those of an application specific nature such as high water mark maintenance and functional integrity.

It is useful to remember that these operation requirements schemas are defining a change in the state of the system from an initial state, *s*, to a final state, *s'*. These definitions are generally included into the schema by the inclusion of the *Trusted_Func*, or *Functional_Integrity* axiom from section 3, which themselves use the definition of a *TRANSITION* from the model in Annex B. The entities and attributes of interest to the particular operation specification are also identified in the signature of the schema. The predicates of these operation schemas define how the various entity-attribute functions and relations comprising a *STATE* are altered by the transition. In order for the specification to be deterministic (which is desirable for a specification of security properties) the Z notation insists that all aspects of the state change are defined, and not only those concerned with the currently interesting entities. In other words, the operation specifications describe the changes that take place in the state, and also what does not change.

Following the specification of the state change caused by the required operation, the associated transition request is specified. In general this uses the same name as operation specification, but without the '_functionality' suffix. This schema identifies those entites that were the *requestors* of the transition, together with those entities whose functionality attributes contributed to the state changes, ie the *observed* entities. The specification of the desired functionality is then conjoined (anded) with the security axioms on a transition, *Security*, as defined in the policy model of Annex B, and some implications discussed.

Note that each of the following sections is an independently typechecked module of Z, and the final part of each section is the *keeps* definition which exports the secure operation specification.

4.1. Useful Schemas

This section defines some general schemas that are used in operation specifications.

policy_model :Module
Defines
STATE, E

Functionality :Module
Defines
DOCUMENT, JOURNAL, USER, user_data, WINDOW, CUPBOARD, DRAFT, TEXT, EVENT

Several of the operations only alter the functionality aspects of the state, ie make no modifications to the control attributes of any entities. Thus the following schema, $\Xi CONTROLS$, defines the state change which does not alter any of the control functions and relations, ie only modifications to the *Other* relation are permitted.

ΞCONTROLS

$s, s' : STATE$

$s'.Class = s.Class$
$s'.Trust = s.Trust$
$s'.Role = s.Role$
$s'.Id = s.Id$
$s'.Conflict = s.Conflict$
$s'.Ref = s.Ref$

Operations which record events in journals will first need to identify the particular journal. The following schemas identify the journal from a document entity and the journal from the *USER* attribute of a window. Note that the first schema insists that the document has exactly one reference to a journal amongst its functionality attributes, and the second that there is exactly one *USER* attribute in the window. In other words, documents must have only one journal, and windows must only belong to one user.

identify_document_journal

$s : STATE$
$document : DOCUMENT$
$doc\_journal : JOURNAL$

$\{ s.Ref(doc\_journal) \} = s.Other(\{document\}) \cap s.Ref(JOURNAL)$

identify_user_journal

$s : STATE$
$window : WINDOW$
$user\_journal : JOURNAL$

$\exists_1 user : USER \mid user \in s.Other(\{window\}) \bullet s.Ref(user\_journal) = (user\_data(user)).journal$

The cupboard entity from the *USER* attribute associated with a window may be identified in the same way as the journal. Again, the schema requires there to be only one *USER* attribute associated with the window.

identify_user_cupboard

$s : STATE$
$window : WINDOW$
$cupboard : CUPBOARD$

$\exists_1 user : USER \mid user \in s.Other(\!|\{window\}|\!) \bullet s.Ref(\ cupboard\ ) = (user\_data(\ user\ )).cupboard$

The operations involving windows, drafts and documents generally require the ability to identify the *TEXT* or *USER* attribute from amongst the general functionality attributes. It is therefore useful to define *User* and *Text* functions as subsets of the general functionality relation, in much the same way as high water marks and flow information were identified in section 3. Note that this is merely a convenience and the same properties may be ensured using the $\exists_1 \mid \bullet$ construct in the operation specifications. Thus, *User* is the subset of functionality concerned with the *USER* attributes of the *WINDOW* entities, and *Text* is concerned with the *TEXT* attribute of windows, drafts and documents. The fact that these are functions requires that each entity has only one of that particular type of attribute.

Useful_Functions

$s : STATE$
$User : E \nrightarrow USER$
$Text : E \nrightarrow TEXT$

$User = WINDOW \lhd s.Other \rhd USER$
$Text = (\ WINDOW \cup DRAFT \cup DOCUMENT\ ) \lhd s.Other \rhd TEXT$

At the level of abstraction of this specification, there are no constraints about the particular *EVENT* attributes that are used to modify journals. It is therefore convenient in an operation specification to be able to simply identify the set of journals that are updated, *update_journals* below, and leave the choice of attribute to a lower level of abstraction (except to the extent that it must obviously be governed by the security axioms).

Update_Journals

$s, s' : STATE$
$update\_journals : \mathbb{P}\ JOURNAL$

$\forall journal : update\_journals \bullet$
$\quad \exists event : EVENT \bullet s'.Other(\!|\{journal\}|\!) = s.Other(\!|\{journal\}|\!) \cup \{\ event\ \}$

Note that the above technique would be less 'clean' to specify if journals ever had high water marks or FLOW attributes. This is because then the total change to the functionality attributes of the journal would not simply be the addition of an *EVENT* attribute.

useful keeps identify_document_journal, identify_user_journal, identify_user_cupboard,
ΞCONTROLS, Useful_Functions, Update_Journals

4.2. Login

policy_model :Module	*Functionality :Module*	*useful :Module*
Defines	Defines	Defines
ID, CLASS, entities,	*Functional_Integrity,*	*Update_Journals*
R, faithful,	*PASSWORD, JOURNAL,*	
dont_signal, creator,	*WINDOW, LOGIN, USER,*	
Security	*USERDATA, user_data,*	
	bottom, owner_user,	
	owner_sso, blank_text,	
	class_of, unclassified,	
	Trusted_Func, no_alteration	

As discussed briefly in section 3.9, a system of separation of duty is used to ensure that only authorised users can login to SERCUS. Logging in is modelled as two transitions, the first of which creates a new window to represent the user and the second upgrades it to the appropriate clearance and degrees of trust. Note that there is no 'confidentiality' problem with modelling login as a single transition where a window is simply created at the appropriate level and with all the necessary attributes. However, the sequence of transitions is modelling the fact that the trusted login software must only log people in in response to a request, and that people can only login if authenticated by the login software.

Creating a window

In SERCUS, a request to login is valid if an authorised *uid* and *password* are presented, ie there is an appropriate *USER* attribute amongst the functionality attributes of the *LOGIN* entity. The clearance of the user and their journal are also recovered from this information so that the window of the user is given the appropriate classification and the journal may be updated to record the login. This is all part of the *user_data* representation of a *USER* attribute.

login_request

uid : ID

password : PASSWORD

authenticate_details

login_request

Functional_Integrity

user : USER

data : USERDATA

user_journal : JOURNAL

clearance : CLASS

---

user ∈ *Appl*({*LOGIN*})

user_data(user) = data

uid = data.uid

password = data.password

s.Ref(user_journal) = data.journal

clearance = data.clearance

In response to such a valid request a new *WINDOW* entity is created. This new window is given the role and id of the particular user and a classification of *bottom* (ie as yet unable to observe the contents of any entities in SERCUS). The window is also given a high water mark attribute. The login is recorded in the user's journal. The conflict of the window is set to allow the particular user (who will be either a security officer or an ordinary user) and the system owners proxy (login software) to upgrade it to the user's clearance. The window therefore needs to be given *faithful* and *dont_signal TRUST* attributes so that it may participate in the subsequent upgrade. This can

be considered to be representing the fact that the software used by the human users to authenticate,etc, themselves must be trusted to act on their behalf only.

Both the *LOGIN* entity and new window are modified with the appropriate flow information for the subsequent transition to raise the classification of the window to the users clearance. This flow information will probably represent the reference to the new window and the clearance to give it. However, as discussed in section 3.11 and section 5, the choice of necessary sequencing information is left to be determined by an implementation, so long as it is only derived from the nominated observed entities and unclassified control attributes.

create_window_functionality

$$
\begin{array}{l}
\textit{Functional\_Integrity} \\
\textit{Update\_Journals} \\
\textit{authenticate\_details} \\
\textit{new\_window} : \textit{WINDOW} \\
\hline
\textit{new\_window} \notin \textit{entities}\ s \\
\exists\ \textit{new\_ref} : \textit{REF} \bullet s'.\textit{Ref} = s.\textit{Ref} \cup \{\ \textit{new\_window} \mapsto \textit{new\_ref}\ \} \\
s'.\textit{Class} = s.\textit{Class} \cup \{\ \textit{new\_window} \mapsto \textit{bottom}\ \} \\
s'.\textit{Trust} = s.\textit{Trust} \cup \{\ \textit{new\_window} \mapsto \textit{faithful}, \textit{new\_window} \mapsto \textit{dont\_signal}\ \} \\
s'.\textit{Role} = s.\textit{Role} \cup \{\ \textit{new\_window} \mapsto \textit{data.role}\ \} \\
s'.\textit{Id} = s.\textit{Id} \cup \{\ \textit{new\_window} \mapsto \textit{uid}\ \} \\
s'.\textit{Conflict}[\![\ \{\textit{new\_window}\}\ ]\!] = \{\ \textit{owner\_user}, \textit{owner\_sso}\ \} \\
\{\ \textit{new\_window}\ \} \lhd s'.\textit{Conflict} = s.\textit{Conflict} \\
\textit{Appl}'[\![\ \{\textit{new\_window}\}\ ]\!] = \{\ \textit{user}, \textit{blank\_text}\ \} \\
\{\ \textit{user\_journal}, \textit{new\_window}\ \} \lhd \textit{Appl}' = \{\ \textit{user\_journal}\ \} \lhd \textit{Appl} \\
\textit{class\_of}(\ \textit{Hwm}'(\ \textit{new\_window}\ )\ ) = \textit{unclassified} \\
\textit{update\_journals} = \{\ \textit{user\_journal}\ \} \\
\textit{changed\_flows} = \{\ \textit{LOGIN}, \textit{new\_window}\ \}
\end{array}
$$

This transition is modelled as requested by the LOGIN entity, which is also the only entity whose attributes influence the outcome of the transition.

create_window

$$
\begin{array}{l}
r? : R \\
\textit{create\_window\_functionality} \\
\hline
r?.\textit{requestors} = r?.\textit{observed} = \{\ \textit{LOGIN}\ \}
\end{array}
$$

Note that this first transition only has a single requestor as there are initially no active entities in the system to represent the user. Because an entity is created, the *LOGIN* entity must possess *creator* trust. Essentially this means that it is trusted to correctly check the uid and password and to setup the appropriate controls on the new window. The *LOGIN* entity must itself be classified *bottom* otherwise the *no_flows_down* axiom would prevent the above transition taking place. This models the fact that the purpose of the login software is to ensure that only authorised users may use the system and that there is no requirement for it to observe any additional information. As mentioned eariler, the protection of the attributes of the *LOGIN* entity is an integrity matter and requires that no inappropriate transitions are specified.

Raising to Clearance

This second transition raises the classification of the new window to the clearance of the user, changes the conflicts to prohibit further alterations to the controls and gives it the remaining *TRUST* attribute, ie *creator* trust. Thus, when they log in, the users of SERCUS are immediately on the Trusted Path.

raise_level_functionality

$Trusted\_Func$
$clearance : CLASS$
$new\_window : WINDOW$

$s'.Class = s.Class \oplus \{ new\_window \mapsto clearance \}$
$s'.Trust = s.Trust \cup \{ new\_window \mapsto creator \}$
$s'.Role = s.Role$
$s'.Ref = s.Ref$
$s'.Id = s.Id$
$s'.Conflict(\{new\_window\}) = \{ no\_alteration \}$
$\{ new\_window \} \lhd s'.Conflict = \{ new\_window \} \lhd s.Conflict$
$Func' = Func$

This transition is viewed as requested by the new window, which is acting on behalf of the user who wishes to log in, and the *LOGIN* entity acting on behalf of the system owners. Both these entities are also observed so that they may discover appropriate information, eg the particular clearance, from the flow information placed in them by the previous transition.

raise_level

$r? : R$
raise_level_functionality

$r?.requestors = r?.observed = \{ new\_window, LOGIN \}$

Note that both requestors must be faithfully acting on behalf of the user and owners respectively. In addition, they must not be using the changes of control, ie the fact that someone has logged in, to signal any information. However, note that both entities are initially classified *bottom* and therefore have no classified information to signal. The integrity requirement is that they must also be trusted not to signal password information, etc.

So securely logging in is the secure creation of a window followed by the secure upgrade of its controls. Note that the property of schema composition, ⨾, which ensures that the window, clearance, etc, identified in the schemas are the same, also requires the requests, *r?*, to be renamed, *r1?*, *r2?*, (using the subscript notation [*new/old*]), otherwise there would only be a single request with contradictory definitions of the requesting and observed entities.

$Login \triangleq (create\_window_{[r1?/r?]} \wedge Security_{[r1?/r?]}) \fatsemi (raise\_level_{[r2?/r?]} \wedge Security_{[r2?/r?]})$

Login keeps Login

4.3. Logout

policy_model :Module	*Functionality :Module*	*useful :Module*
Defines	Defines	Defines
R, Security	*Trusted_Func, WINDOW*	*Update_Journals, identify_user_journal*

Users request to logout from one of the windows of their display. The effect is that all the windows of that user, ie they have the same ID control attribute, are removed from all the relations and functions comprising the state. The event is recorded in the user's journal.

logout_functionality

Trusted_Func
Update_Journals
window : WINDOW
identify_user_journal

$s'.Class = destroyed \lhd s.Class$
$s'.Trust = destroyed \lhd s.Trust$
$s'.Role = destroyed \lhd s.Role$
$s'.Id = destroyed \lhd s.Id$
$s'.Conflict = destroyed \lhd s.Conflict$
$s'.Ref = destroyed \lhd s.Ref$
$\{ user\_journal \} \lhd Func' = (destroyed \cup \{user\_journal\}) \lhd Func$

where

$destroyed == \{ w : WINDOW \mid sId(w) = sId(window) \}$

$update\_journals = \{ user\_journal \}$

The requestor of this transition is the particular window, and this is also the only entity whose functionality attributes were observed. Note that since the Id of an entity is a control and the model considers all controls to be visible and unclassified, no functionality attributes need to be observed to find the appropriate windows to destroy.

logout

$r? : R$
logout_functionality

$r?.requestors = r?.observed = \{ window \}$

So the secure transition is

$Logout \triangleq logout \wedge Security$

The *no_signalling* axiom requires that the requesting window possesses the *dont_signal* trust attribute, ie logging out should not be requested by any untrusted software the use may be running.

Logout keeps Logout

4.4. Creating an Untrusted Window

policy_model :Module	*Functionality :Module*	*useful :Module*
Defines	Defines	Defines
E, REF, entities,	*Trusted_Func, WINDOW,*	*Useful_Functions*
R, Security	*no_alteration, blank_text,*	
	class_of, unclassified	

Once logged in, further windows may be created. Some of these will be further Trusted Path windows and are not particularly interesting to specify as the window essentially creates a copy of itself. Other new windows model the activation of untrusted software, such as a word processing package, and consequently these windows will not be given any *TRUST* attributes. All new windows in SERCUS inherit the clearance, role and id of the user they are representing, and will also be given the same *USER* attribute as part of their functionality. SERCUS does not require the creation of a new window to be journalled. Also, the requesting window is modified with the reference to the new window.

The requirement for new untrusted windows is that they are initially given blank text and an unclassified high water mark. However, in cases where the window that requests the creation of a new window does not have an unclassified high water mark itself, the *Trusted_Func* axiom requires that the *evaluator* role agrees that this 'downgrade' is acceptable. In this case the justification is that in all circumstances the new window is given the same attributes. Consequently, no matter what classification of information the requesting window has accessed it cannot encode it in the contents of the new window. Thus there is a second requestor to this transition, namely an entity acting on behalf of the evaluator.

create_untrusted_functionality

$$
\begin{array}{l}
Trusted\_Func \\
Useful\_Functions \\
window : WINDOW \\
evaluator\_entity : E \\
\hline
\exists\, new\_window : WINDOW;\ new\_ref : REF \bullet \\
\quad new\_window \notin entities\ s \\
\quad s'.Class = s.Class \cup \{ new\_window \mapsto s.Class(window) \} \\
\quad s'.Role = s.Role \cup \{ new\_window \mapsto s.Role(window) \} \\
\quad s'.Id = s.Id \cup \{ new\_window \mapsto s.Id(window) \} \\
\quad s'.Ref = s.Ref \cup \{ new\_window \mapsto new\_ref \} \\
\quad s'.Trust = s.Trust \\
\quad s'.Conflict(\{new\_window\}) = \{ no\_alteration \} \\
\quad \{new\_window\} \lhd s'.Conflict = s.Conflict \\
\quad Func' = Func \cup \{ window \mapsto new\_ref, \\
\qquad\qquad new\_window \mapsto blank\_text, \\
\qquad\qquad new\_window \mapsto User(window) \} \\
\quad class\_of(Hwm'(new\_window)) = unclassified
\end{array}
$$

create_untrusted

$$
\begin{array}{l}
r? : R \\
create\_untrusted\_functionality \\
\hline
r?.requestors = \{ window, evaluator\_entity \} \\
r?.observed = \{ window \}
\end{array}
$$

So the secure operation is,

CreateUntrusted ≙ *create_untrusted* ∧ *Security*

The *no_signalling* axiom requires that the original window be trusted not to write or encode classified information into the existence and controls of the new window, ie the requestors possess the *dont_signal TRUST* attribute. They also require *creator* trust. Note that, as discussed above, the use of the evaluator as a second requestor simply means that the code to create new windows is also trusted not to include classified information in the contents attributes of the new window. The use of the evaluator role is further discussed in section 5.

CreateUntrusted keeps CreateUntrusted

4.5. Creating a Draft Document

policy_model :Module	*Functionality :Module*	*useful :Module*
Defines *E, REF, R, entities, Security*	Defines *Trusted_Func, WINDOW, DRAFT, add, class_of, unclassified, hwm_user, hwm_sso*	Defines *Useful_Functions*

Drafts are documents that have not yet been given their final classification and made available to all users by being placed in the CDR. The requirement is that, since users may type in, or include by cut and paste operations, information up to their clearance, the draft be classified at this level. However, it is also required that users be able to create documents lower than their clearance, and more importantly, be able to edit them using off-the-shelf wordprocessing packages, ie untrusted software. Therefore, a high water mark reflecting the classification of information included into the draft is maintained. A document may be created, from the draft, at a level lower than the user's clearance, but not lower than this high water mark.

The request to create a new draft entity is made by a window. The functionality requirement is that the new draft initially contains blank text with an associated unclassified high water mark. Therefore, as for creating an untrusted window, the *Trusted_Func* axiom requires that an evaluator agrees that this is acceptable, even when the requesting window has observed classified information, ie has a high water mark greater than unclassified. In order to ensure the functionality requirement that only the originator of a draft may access it in any way, drafts are labelled with the ID of the user who created them, and this is checked for the edit and document creation operations. The separation of duty controls on the new draft are set up to allow the originator (who will be either a security officer or an ordinary user) and the trusted high water mark code to downgrade it to the required level of the final document. Since draft documents are private to a particular user, SERCUS does not require that their creation is journalled. The reference to the new draft document is added to the text of the requesting window.

create_draft_functionality

$$
\begin{array}{l}
Trusted\_Func \\
Useful\_Functions \\
window : WINDOW \\
evaluator\_entity : E \\
\hline
\exists\, new\_draft : DRAFT;\ new\_ref : REF; \\
\quad new\_text : add(\!|\ \{(\ Text(\ window\ ),\ \{new\_ref\}\ )\}\ |\!) \bullet \\
\qquad new\_draft \notin entities\ s \\
\qquad Func' = Func \setminus \{\ window \mapsto Text(\ window\ )\ \} \\
\qquad\qquad \cup \{\ window \mapsto new\_text,\ new\_draft \mapsto blank\_text\ \} \\
\qquad class\_of(\ Hwm'(\ new\_draft\ )\ ) = unclassified \\
\qquad s'.Class = s.Class \cup \{\ new\_draft \mapsto s.Class(\ window\ )\ \} \\
\qquad s'.Id = s.Id \cup \{\ new\_draft \mapsto s.Id(\ window\ )\ \} \\
\qquad s'.Conflict(\!|\ \{new\_draft\}\ |\!) = \{\ hwm\_user,\ hwm\_sso\ \} \\
\qquad \{new\_draft\} \lhd s'.Conflict = s.Conflict \\
\qquad s'.Ref = s.Ref \cup \{\ new\_draft \mapsto new\_ref\ \} \\
s'.Trust = s.Trust \\
s'.Role = s.Role
\end{array}
$$

create_draft

$r? : R$

create_draft_functionality

$r?.requestors = \{ window, evaluator\_entity \}$

$r?.observed = \{ window \}$

The secure operation is the combination of the above functionality with security.

$CreateDraft \triangleq create\_draft \wedge Security$

The *no_signalling* axiom requires that the code be trusted not to write or encode classified information into the existence and controls of the new draft, ie the requestors possess the *dont_signal TRUST* attribute. The basis of this trust could be that the requesting window is part of the Trusted Path, ie the human user requested the operation. Alternatively, the evaluator could agree that new drafts could be created by untrusted software if they were convinced that no other window could discover the existence of another's drafts. Note that the use of the evaluator as a second requestor simply means that the code to create new windows is trusted not to write or otherwise encode classified information in the contents attributes of the new draft.

CreateDraft keeps CreateDraft

4.6. Editing a Draft

policy_model :Module	*Functionality :Module*	*useful :Module*
Defines	Defines	Defines
R, Security	*Trusted_Func, WINDOW, DRAFT, merge*	*Useful_Function, ΞCONTROLS*

Editing a draft document using an untrusted word processing package can be modelled by the following transitions which give the untrusted software some text to edit, and then replace the edited text into the draft entity. In between these two transitions the untrusted software may have called any number of other transitions, for example opening a document, and the high water mark monitors the classification of the information included in the edit by this means.

Starting the Edit
Starting the untrusted edit is modelled as requested by a window. This window takes the text from a draft entity, and passes this text to an untrusted window (representing the active word processing software). The *Trusted_Func* axiom raises the high water mark of the untrusted window according to the high water mark of the draft. Note that a functional integrity requirement is that only the orginator of a draft may access it, and therefore it is required that the *ID* attributes of both windows and the draft entity be representing the same human user.

start_edit_functionality

$$
\begin{array}{l}
Trusted\_Func \\
Useful\_Functions \\
\Xi CONTROLS \\
window, untrusted : WINDOW \\
draft : DRAFT \\
\hline
sId(window) = sId(untrusted) = sId(draft) \\
\exists\, edit\_text : merge[\![\{\{ Text(draft), Text(untrusted) \}\}]\!] \bullet \\
\quad Func' = Func \setminus \{ untrusted \mapsto Text(untrusted) \} \\
\qquad\qquad \cup \{ untrusted \mapsto edit\_text \}
\end{array}
$$

Both the windows and the draft are observed by this transition.

start_edit

$$
\begin{array}{l}
r? : R \\
start\_edit\_functionality \\
\hline
r?.requestors = \{ window \} \\
r?.observed = \{ untrusted, draft, window \}
\end{array}
$$

The secure transition is the combination of the above functionality with the security axioms.

$StartEdit \triangleq start\_edit \wedge Security$

Since no entities are created or destroyed by this transition, and no controls altered, there is no requirement for the requesting window to possess any *TRUST* attributes. The only security constraint is imposed by *no_flows_down*. However, both windows and the draft will be at the same classification level, ie clearance of the user, anyway.

Finishing the Edit
The end of an edit is modelled as requested by a window. The functionality is that the text of the draft entity is replaced with the text from the untrusted window. As above, the *Trusted_Func* axiom ensures that the high water mark is raised appropriately.

stop_edit_functionality

$Trusted\_Func$
$Useful\_Functions$
$\Xi CONTROLS$
$window, untrusted : WINDOW$
$draft : DRAFT$

$sId(window) = sId(untrusted) = sId(draft)$
$Func' = Func \setminus \{ draft \mapsto Text(draft) \} \cup \{ draft \mapsto Text(untrusted) \}$

stop_edit

$r? : R$
$stop\_edit\_functionality$

$r?.requestors = \{ window \}$
$r?.observed = \{ untrusted, draft, window \}$

The secure transition is the combination of the above functionality with the security axioms.

$StopEdit \triangleq stop\_edit \wedge Security$

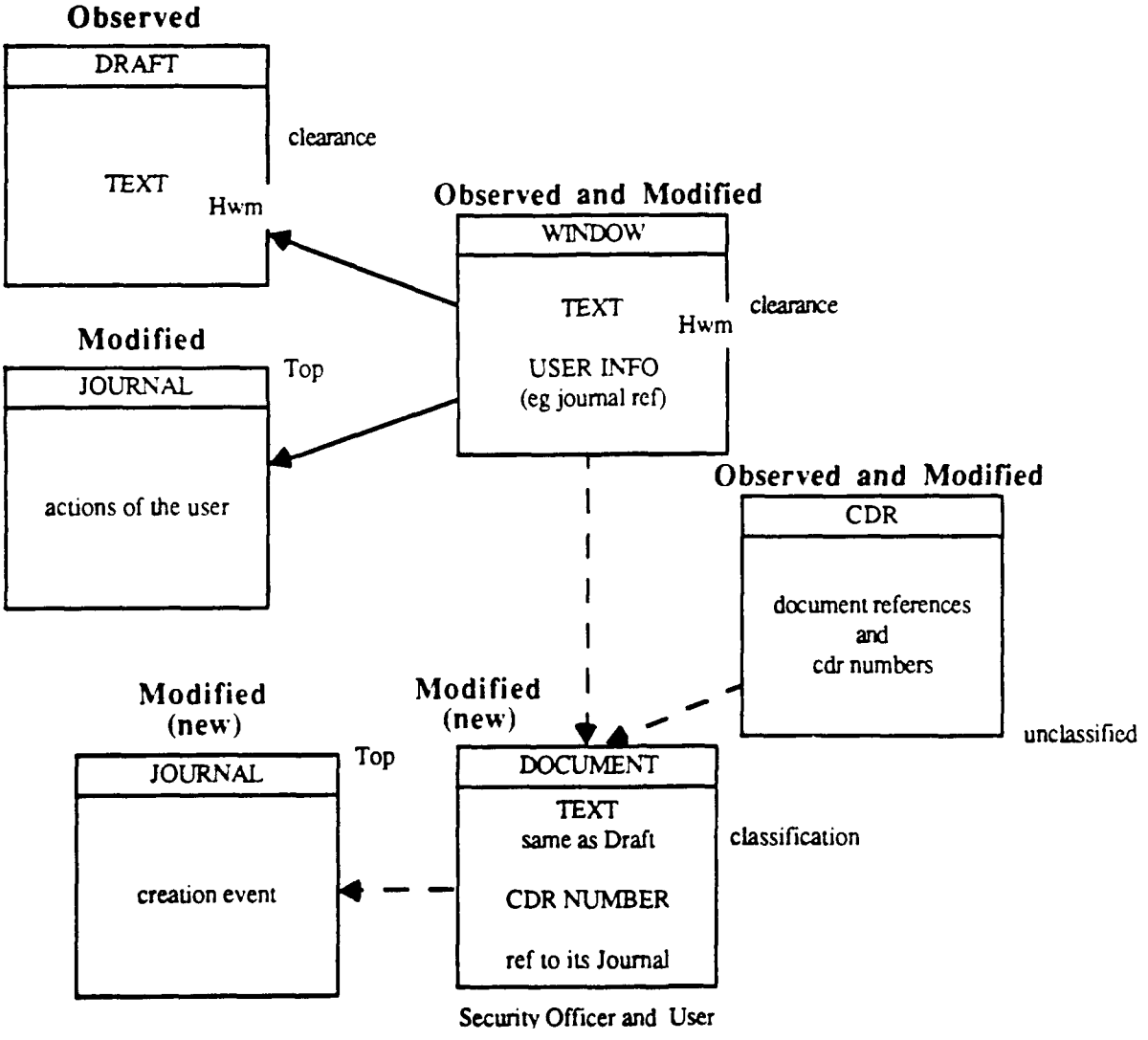
As above, there is no necessity for the requesting window to possess any *TRUST* attributes.

EditDraft keeps StartEdit, StopEdit

4.7. Creating a Document

Users of SERCUS may create new classified documents from their drafts. Essentially, the requirement is that the text of the draft is copied into a new document, and the document is given the next available CDR number and a new journal, initially recording the creation event. A further parameter to the operation is the required classification for the document. However, to prevent the new document from being underclassified, this classification must not be lower that the high water mark of the draft. Additional requirements are that users may not create documents higher than their clearance and that any references in the text of the new document are only for existing documents, and not to any other type of object. The reference to the new document is added to the text of the window from which the user requested the operation. It is also required that the new document is added to the central list of documents, ie the CDR is updated with the document reference and associated CDR number. In addition, the creation event is recorded in the journal of the user who requested the operation. The following diagram, Figure 4, illustrates this, and indicates which entities were observed and which modified by the creation operation.

Figure 4: Creating a new Document from a Draft



This requirement cannot be modelled as a single transition for the following important reasons:

a) The draft document is observed to copy the text and hence the *no_flows_down* axiom would require that the new document could not be classified lower.

b) The greatest lower bound of the modified entities is unclassified (the CDR) and the least upper bound of the observed is the clearance of the user requesting the operation. Consequently, the *no_flows_down* axiom would require that the clearance of the user when creating a document be

no higher than unclassified, and consequently documents could not be created higher than unclassified.

This is not the desired functionality.

Thus, the model has highlighted the downward flows of information that occur from the requirement to create a document. Consequently, creating a document needs to be modelled as a sequence of individually secure transitions and the necessary sequencing information passed between them using the *FLOW* attributes outlined in section 3.11. There are possibly as many sequences to choose from as there are people to model them. However, all of these sequences will require that the information in the draft and the update to the CDR are downgraded at some point. For instance, the draft can either be downgraded before the document is created from it or the document created and then downgraded. There are no 'confidentiality' considerations concerning when the update to the user's journal takes place. However, it must be remembered that if the implementation of the *EVENT* attributes decides to include the CDR number and/or document reference, this has to be allowed for in the sequencing.

policy_model :Module	*Functionality :Module*	*useful :Module*
Defines	Defines	Defines
E, REF, CLASS, >=, R, dont_signal, creator, entities, Security	*Trusted_Func, WINDOW, DRAFT, refs_of, DOCUMENT, Functional_Integrity, CDR, the_evaluator, newer, entry, Journal, top, no_alteration, sso_user,unclassified, add*	*Useful_Functions, Update_Journals, identify_user_journal, ΞCONTROLS*

Creating a document from a draft document can be modelled by the following sequence of individually secure transitions.

Downgrade Draft

The confidentiality controls, ie *no_flows_down*, prevent the contents of the draft entity (protected at the clearance of the user) from being copied into a new document entity classified lower. Therefore, the first transition of the document creation sequence, is the downgrade of the draft entity to the required classification, so long as this is between the draft's high water mark and the clearance of the user. This downgrade is justified by the separation of duty between the owner of the draft and the trusted high water mark software. Also note that this may only be performed by the user who created the draft.

downgrade_draft_functionality

$Trusted\_Func$
$Useful\_Functions$
$window : WINDOW$
$hwm\_man : E$
$draft : DRAFT$
$required\_class : CLASS$

$sId(draft) = sId(window)$
$required\_class >= Hwm(draft)$
$s.Class(window) >= required\_class$
$s'.Class = s.Class \oplus \{ draft \mapsto required\_class \}$
$s'.Trust = s.Trust$
$s'.Role = s.Role$
$s'.Id = s.Id$
$s'.Conflict = s.Conflict$
$s'.Ref = s.Ref$
$Func' = Func$

This transition is requested by both a window entity and an entity representing the high water mark software. Both the window and draft may be observed. Also note that no entities are *modified* by this transition, as the only alteration to the state is the new classification attribute given to the draft entity. Therefore, as no entities or attributes are created there is no requirement to provide other transitions in the sequence with *FLOW* attributes.

downgrade_draft

$r? : R$

downgrade_draft_functionality

$r?.requestors = \{ window, hwm\_man \}$

$r?.observed = \{ draft, window \}$

Create New Document

The confidentiality controls, ie *no_flows_down*, prevent an observed window from creating an entity classified lower than itself, and therefore the window still cannot create a new document from the downgraded draft. It is also not appropriate to downgrade the window as its attributes will not then be adequately protected. Therefore, the window creates a worker entity and gives it the appropriate information, modelled by a *FLOW* attribute, which will contain sufficient information to enable the worker to perform the appropriate actions with the draft. The worker has to be classified at the level of the window since its flow information is given to it from the window. However, its controls are such that it may be downgraded to the classification of the draft by an entity with the evaluator role in a subsequent transition.

create_worker_functionality

Functional_Integrity

$window : WINDOW$

identify_user_journal

$worker : E$

$worker \notin entities\ s$

$s'.Class = s.Class \cup \{ worker \mapsto s.Class(window) \}$

$s'.Trust = s.Trust$

$s'.Role = s.Role$

$s'.Id = s.Id$

$s'.Conflict = s.Conflict \cup \{ worker \mapsto the\_evaluator \}$

$\exists new\_ref : REF \bullet s'.Ref = s.Ref \cup \{ worker \mapsto new\_ref \}$

$Appl' = Appl$

$changed\_flows = \{ worker \}$

This transition is requested by the window entity, which is also the only observed entity.

create_worker

$r? : R$

create_worker_functionality

$r?.requestors = r?.observed = \{ window \}$

The human evaluator, represented by an entity which is neither observed nor modified (see Section 5, discussion), then agrees that the actions of the worker are acceptable and downgrades it to the level of the draft, also giving it the necessary *TRUST* attributes. No entities need to be modified with any flow information as no entities nor attributes are created and the worker entity was given the appropriate information in the previous transition.

downgrade_worker_functionality

Trusted_Func
evaluator_entity, worker : E
draft : DRAFT

$s'.Class = s.Class \oplus \{ worker \mapsto s.Class(draft) \}$
$s'.Trust = s.Trust \cup \{ worker \mapsto dont\_signal, worker \mapsto creator \}$
$s'.Role = s.Role$
$s'.Id = s.Id$
$s'.Conflict = s.Conflict$
$s'.Ref = s.Ref$
$Func' = Func$

downgrade_worker

r? : R
downgrade_worker_functionality

$r?.requestors = \{ evaluator\_entity \}$
$r?.observed = \{ worker \}$

The next available CDR number for the document is identified, by the *next_cdr_number* schema.

next_cdr_number

Trusted_Func
new_cdr_num : CDR_NUM

$new\_cdr\_num \notin used\_numbers$
$\forall c : unused\_numbers \bullet c$ *newer* new_cdr_num
where
$used\_numbers == (Func \fatsemi entry \fatsemi fst)(\!| \{CDR\} |\!)$
$unused\_numbers == CDR\_NUM \setminus used\_numbers$

The worker then creates new document and journal entities, and sets up the controls so that the journal can never be regraded and the document may only be regraded by agreement between a user and a security officer. The text of the draft is checked to ensure it only contains references to documents and is copied into the document, along with the new cdr number and reference to the new journal. Appropriate events are added to the user's journal and the journal of the document. Functional integrity requires that both the worker and the window are updated with sequencing information, ie *FLOW* attributes.

create_document_functionality

Functional_Integrity
Useful_Functions
Update_Journals
worker : E
window : WINDOW
draft : DRAFT
next_cdr_number
document : DOCUMENT
journal, user_journal : JOURNAL
doc_ref, journal_ref : REF

$\{ document, journal \} \cap entities\ s = \{\}$
$s'.Ref = s.Ref \cup \{ document \mapsto doc\_ref, journal \mapsto journal\_ref \}$
$s'.Class = s.Class \cup \{ document \mapsto s.Class(worker), journal \mapsto top \}$
$s'.Conflict[\![\{journal\}]\!] = \{ no\_alteration \}$
$s'.Conflict[\![\{document\}]\!] = \{ sso\_user \}$
$\{ document, journal \} \lhd s'.Conflict = s.Conflict$
$refs\_of(Text(draft)) \subseteq s.Ref[\![DOCUMENT]\!]$
$Appl'[\![\{document\}]\!] = \{ Text(draft), new\_cdr\_num, journal\_ref \}$
$\{ document, journal, user\_journal \} \lhd Appl' = Appl$
$update\_journals = \{ journal, user\_journal \}$
$s'.Trust = s.Trust$
$s'.Role = s.Role$
$s'.Id = s.Id$
$changed\_flows = \{ worker, window \}$

This transition is requested by the worker entity which observes itself, the draft and the CDR.

create_document

r? : R
create_document_functionality

$r?.requestors = \{ worker \}$
$r?.observed = \{ worker, draft, CDR \}$

Note that the *FLOW* attribute given to the window should enable it to update its text with the reference to the new document, although whether the implementation also wishes to supply the window with the new CDR number as well is an application specific issue which is not of interest at this level of specification. The information the worker is modified with should enable it to modify the unclassified CDR with the appropriate information in a later transition. However, although the details of which information the worker and window entities are modified with is not specified, the security axioms require that the implemention may only use information from the nominated observed entities and unclassified control information.

Update CDR

In order to modify the unclassified CDR the worker needs to be downgraded to unclassified. This is again performed by an entity representing the human evaluator.

downgrade_again_functionality

Trusted_Func
evaluator_entity, worker : E

$s'.Class = s.Class \oplus \{ worker \mapsto unclassified \}$
$s'.Trust = s.Trust$
$s'.Role = s.Role$
$s'.Id = s.Id$
$s'.Conflict = s.Conflict$
$s'.Ref = s.Ref$
$Func' = Func$

downgrade_again

r? : R
downgrade_again_functionality

$r?.requestors = \{ evaluator\_entity \}$
$r?.observed = \{ worker \}$

The worker then updates the CDR with information from its FLOW attribute, and is destroyed, ie it is removed from all the functions and relations that comprise the state.

update_cdr_functionality

Trusted_Func
worker : E
new_cdr_num : CDR_NUM
doc_ref : REF

$\exists new\_entry : ENTRY \mid entry(new\_entry) = (new\_cdr\_num, doc\_ref) \bullet$
$Func(\!| \{CDR\} |\!) = Func(\!| \{CDR\} |\!) \cup \{ new\_entry \}$
$\{ CDR \} \lhd Func' = \{ CDR, worker \} \lhd Func$
$s'.Class = \{ worker \} \lhd s.Class$
$s'.Trust = \{ worker \} \lhd s.Trust$
$s'.Role = \{ worker \} \lhd s.Role$
$s'.Conflict = \{ worker \} \lhd s.Conflict$
$s'.Id = \{ worker \} \lhd s.Id$
$s'.Ref = \{ worker \} \lhd s.Ref$

update_cdr

r? : R
update_cdr_functionality

$r?.requestors = \{ worker \}$
$r?.observed = \{ CDR, worker \}$

Update Window
Finally the window of the user is updated with the reference to the new document. Note that when the worker entity created the document it modified the window with appropriate flow information, and consequently the window is able to update itself with the correct reference.

update_window_functionality

$Trusted\_Func$
$Useful\_Functions$
$\Xi CONTROLS$
$window : WINDOW$
$doc\_ref : REF$

$\exists\, new\_text : add(\{(Text(window), \{doc\_ref\})\}) \bullet$
$Func' = Func \setminus \{ window \mapsto Text(window) \} \cup \{window \mapsto new\_text\}$

update_window

$r? : R$
$update\_window\_functionality$

$r?.requestors = r?.observed = \{ window \}$

The complete specification of securely creating a document is therefore the following sequence. Note that the property of schema composition which ensures that the windows, etc, identified in the various schemas are the same, also requires the various requests, *r?*, to be renamed, *r1?*, *r2?*, etc, (using the subscript notation [*new/old*]), otherwise there would only be a single request with contradictory definitions of the requesting and observed entities.

$$
\begin{aligned}
CreateDocument \triangleq\ & (downgrade\_draft_{[r1?/r?]} \wedge Security_{[r1?/r?]}) \\
& \fatsemi (create\_worker_{[r2?/r?]} \wedge Security_{[r2?/r?]}) \\
& \fatsemi (downgrade\_worker_{[r3?/r?]} \wedge Security_{[r3?/r?]}) \\
& \fatsemi (create\_document_{[r4?/r?]} \wedge Security_{[r4?/r?]}) \\
& \fatsemi (downgrade\_again_{[r5?/r?]} \wedge Security_{[r5?/r?]}) \\
& \fatsemi (update\_cdr_{[r6?/r?]} \wedge Security_{[r6?/r?]}) \\
& \fatsemi (update\_window_{[r7?/r?]} \wedge Security_{[r7?/r?]})
\end{aligned}
$$

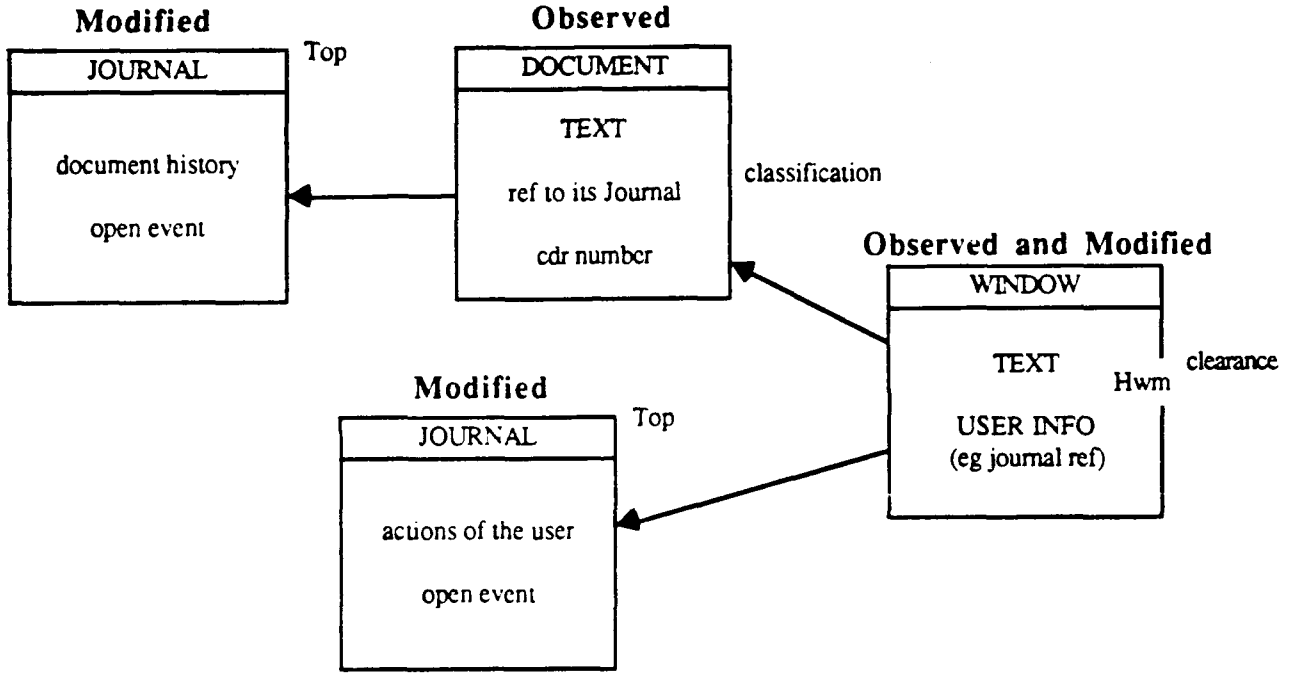
Note that the *no_signalling* and *Separation_of_Duty* axioms require that the window that the operation was requested from be part of the Trusted path, ie possess *faithful* and *dont_signal* trust attributes. Also, in order to be able to create the worker entity, the *Trusted_Creation* axiom requires that it also possesses *creator* trust.

CreateDocument keeps CreateDocument

4.8. Opening a Document

When documents are opened their text becomes available to the requesting window and the event is recorded in both the journal of the particular user and that of the document. The *no_flows_down* security axiom will prevent the window observing documents that it is not cleared for. However, in this case there is an additional functionality requirement that these unsuccessful attempts to open documents should be journalled, as they may indicate that a cleared user has been subverted or was using untrusted software containing a Trojan Horse. Note that these unsuccessful operations are not of interest to the document, and could not be recorded in its journal anyway as the document would have to be observed to discover the appropriate journal to update. Thus, in the unsuccessful case, only the user journal is modified and the window observed. Also note that in the case of the successful open, when the text of the window is merged with the text of the document, the window high water mark may be raised to accommodate the classification of the document. Figure 5 illustrates the successful opening of a document.

Figure 5: Opening a Document



policy_model :Module	*Functionality :Module*	*useful :Module*
Defines *R, >=, Security*	Defines *Trusted_Func, WINDOW, Document, merge*	Defines *Useful_Finctions, Update_Journals, ΞCONTROLS, identify_user_journal, identify_document_journal*

Successful Open

The successful open of the document is modelled as requested by a window. No security controls are altered by the operation. The journal of the user is discovered by observing the requesting window entity to find the *USER* attribute and associated journal, by *identify_user_journal*, and the journal of the document identified by observing the document entity, by *identify_document_journal*, as specified in section 4.1. Also the use of the *Text* function insists that both the document and window only have one *TEXT* attribute each. The window is modified with a new *TEXT* attributes which represents one of the possible ways of merging the two original *TEXT* attributes, as discussed in section 3.4. Events (which could be the same attribute) are added to both journals.

open_document_functionality

Trusted_Func
Useful_Functions
Update_Journals
Ξ*CONTROLS*
window : WINDOW
document : DOCUMENT
identify_user_journal
identify_document_journal

$\exists$ *new_text : merge*[{{ *Text(document), Text(window) }}] •
update_journals ◁ *Func'* = *update_journals* ◁ *Func*
\ { *window* ↦ *Text(window) }* ∪ { *window* ↦ *new_text* }
update_journals = { *doc_journal, user_journal* }

This transition is requested by the window which observes itself and the document entity. The journals are not observed by this operation as their contents will be classified *top*, and therefore their update must be implemented as a pure write. In other words, the window must be unable to detect anything about the contents of the journal, and so, for example, the update operation must never fail even if the journal is full.

succesful_open_document

r? : R
open_document_functionality

---

r?.requestors = { *window* }
r?.observed = { *window, document* }

Unsuccessful Open

The attempt to open a document for which the requestor is not cleared simply records the event in the journal of the user, again found by observing the window entity.

attempted_open_document_functionality

Trusted_Func
Update_Journals
Ξ*CONTROLS*
window : WINDOW
document : DOCUMENT
identify_user_journal

¬ (*s.Class(window) >= s.Class(document)*)
update_journals = { *user_journal* }
update_journals ◁ *Func'* = *update_journals* ◁ *Func*

This transition is requested by the window, which is also the only observed entity. As above, the update to the user's journal must be a pure write.

attempted_open_document

$r? : R$

attempted_open_document_functionality

$r?.requestors = r?.observed = \{ window \}$

Opening a document is therefore the attempted transition unless the open will be successful, ie the precondition of the successful schema is true. So, securely opening a document is

$open\_document \triangleq attempted\_open\_document \oplus succesful\_open\_document$
$OpenDocument \triangleq open\_document \wedge Security$

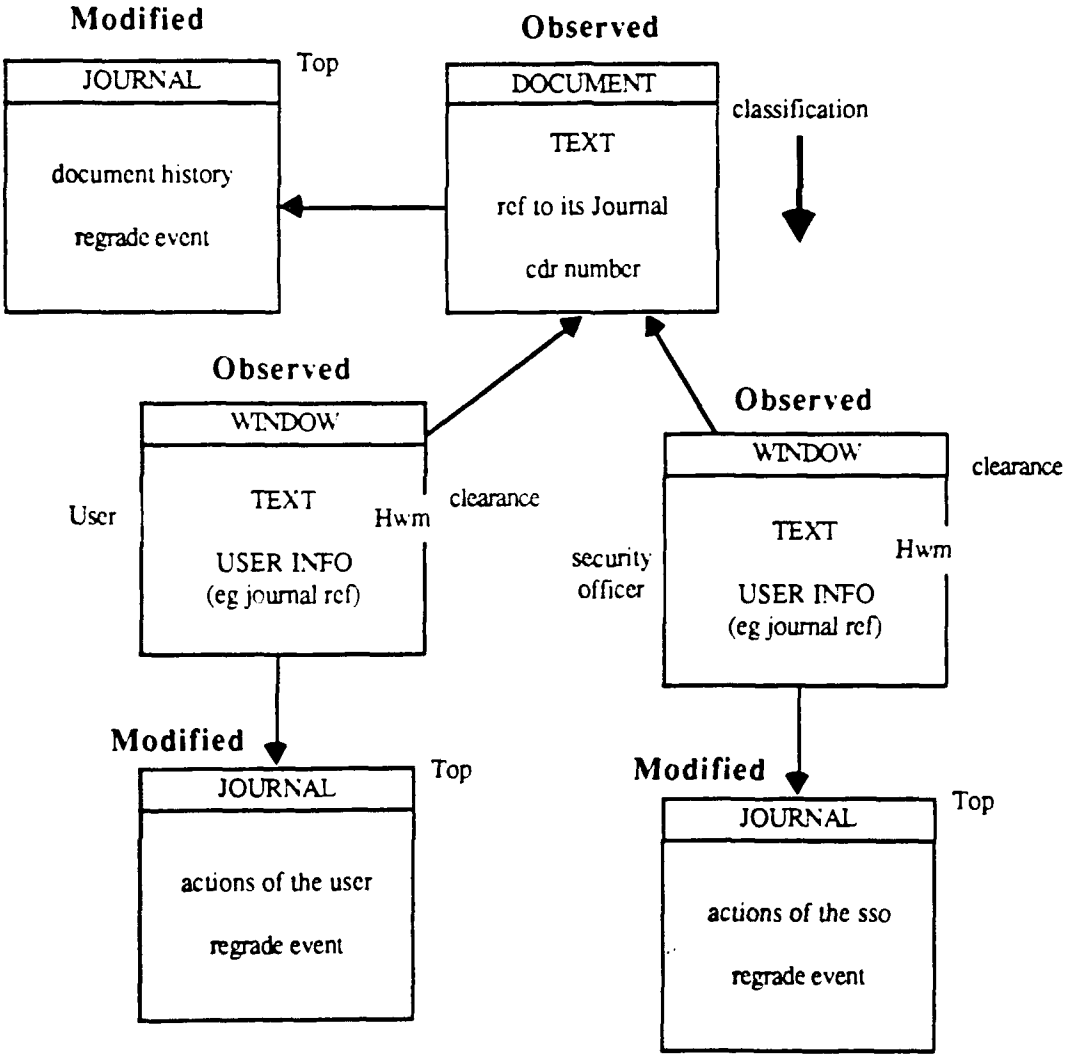
OpenDocument keeps OpenDocument

4.9. Regrading a Document

The separation of duty controls on documents are set up to ensure that a user and a security officer agree that any alterations to the classification (or other controls) of the document are appropriate. Thus, the regrade operation is modelled as being requested by two windows. These windows both observe the text of the document in question and agree the new classification. The journals of both users and the document are updated to record the event.

Note that neither window is modified. This is because if the windows were both *observed* and *modified*, in order to uphold confidentiality the *no_flows_down* axiom would require them to be at the same level, and generally security officers have higher clearances than ordinary users. This illustrates the fact that an observed requestor may potentially encode information in their decision regarding the appropriateness of the transition. Since neither requestor is modified, they will only know their own decision, and not whether the regrade took place. Remember the *dont_signal* trust attribute only ensures that the requestors are not encoding information into the new classification of the document, and not the fact that it has been given a new classification. This could also be considered to be representing the real world situation, where security officers do not remember the particulars of all the regrades they agree to, but are able to look up details in a journal if necessary.

Figure 6: Changing the Classification of a Document



policy_model :Module	*Functionality :Module*	*useful :Module*
Defines	Defines	Defines
CLASS, R, Security	*Trusted_Func, WINDOW, DOCUMENT*	*Update_Journals, identify_user_journal, identify_document_journal*

The regrade operation is requested by two windows, *windowa* and *windowb*. The journals of the users associated with the two windows are discovered by observing the window, as specified in section 4.1 Note that this requires renaming (using the subscript notation [*new/old*]), as the particular schemas define *window* and *user_journal* entities. The journal of the document is discovered by observing the document. Events (which could be the same attribute) are added to all three journals and the classification of the document altered. As discussed above, neither window is modified.

regrade_document_functionality

$Trusted\_Func$
$Update\_Journals$
$windowa, windowb : WINDOW$
$document : DOCUMENT$
$required\_classification : CLASS$
$identify\_user\_journal_{[windowa/window,\ usera\_journal/user\_journal]}$
$identify\_user\_journal_{[windowb/window,\ userb\_journal/user\_journal]}$
$identify\_document\_journal$

$update\_journals = \{ doc\_journal, usera\_journal, userb\_journal \}$
$update\_journals \lhd Func' = update\_journals \lhd Func$
$s'.Class = s.Class \oplus \{ document \mapsto required\_classification \}$
$s'.Trust = s.Trust$
$s'.Role = s.Role$
$s'.Id = s.Id$
$s'.Conflict = s.Conflict$
$s'.Ref = s.Ref$

The regrade transition is modelled as requested by the two windows. The document is observed in order to find the appropriate journal in which to record the event, and to read the text to see if the new classification is appropriate for it. The windows are observed, to discover the users journals and any information such as which document to regrade, or the required classification. However, note that the implementation does not have provide the latter information in the window but could prompt the human user for input.

regrade_document

$r? : R$
$regrade\_document\_functionality$

$r?.requestors = \{ windowa, windowb \}$
$r?.observed = \{ windowa, windowb, document \}$

The secure transition is the combination of the above functionality with the security axioms.

$RegradeDocument \triangleq regrade\_document \wedge Security$

The *Separation_of_Duty* and *no_siganlling* axioms will require that both windows possess *faithful*

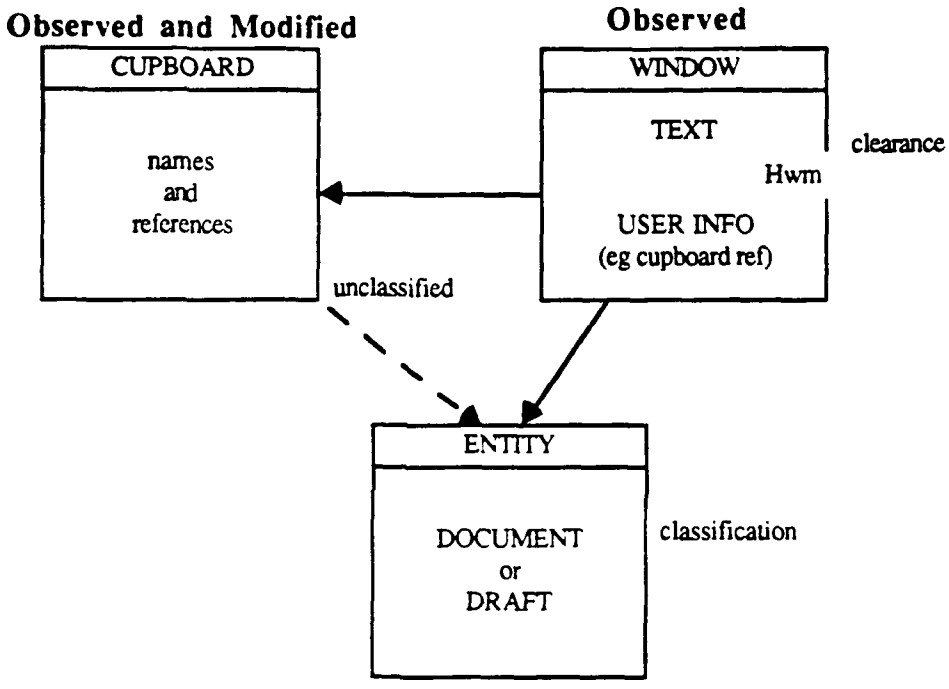
and *dont_signal TRUST* attributes. In other words, both windows must be faithfully acting on behalf of different human users and not be using the change in classification to encode classified information. The *CONFLICT* attribute of the document (as discussed in section 3.1 and specified in the document creation operation) will require that one of these users possess security officer role and the other the ordinary user role. Also note that since neither window is modified, these users may have different clearances, although obviously both must be cleared to observe the contents of the document.

RegradeDocument keeps RegradeDocument

4.10. Keeping in the Cupboard

The users of SERCUS each have a personal cupboard in which they may store documents they are drafting and finished documents. Each item in the cupboard is given a name. The requirement is that the names and references in the cupboard do not convey any classified information, ie in a similar way to the CDR, the cupboard and its contents will be unclassified. To keep something in the cupboard a name and reference are presented. The name must not already exist in the cupboard, and therefore the cupboard needs to be observed. The window the operation was requested from is observed to discover the user's cupboard. SERCUS does not require storing references in the cupboard to be journalled. Figure 7 illustrates the requirement.

Figure 7: Keeping in the Cupboard



This requirement cannot be modelled as a single transition because the cupboard is modified and the window observed. Therefore the confidentiality requirement, *no_flows_down*, would require that references could only be kept in the cupboard by unclassified windows. This is not the desired functionality.

Thus the model has highlighted the potential leakages of information through the names and references chosen to be put into a lowly classified cupboard. Therefore, as for creating a document, the operation to keep a reference in the cupboard can be specified as a sequence of individually secure transitions.

policy_model :Module	*Functionality :Module*	*useful :Module*
Defines *REF, E, entities, R, Security*	Defines *Functional_Integrity, WINDOW, DRAFT, DOCUMENT, NAME, ITEM, the_evaluator, CUPBOARD, item*	Defines *identify_user_cupboard*

Storing an object in an unclassified cupboard requires that a worker entity be created and downgraded by the evaluator, in a similar way that the worker entity was used to update the CDR when a new document was created.

Create Worker

The cupboard of the user is identified from the *USER* attribute of the requesting window, and is observed to check that the name is not already in the cupboard. A new worker entity is created and given an appropriate *FLOW* attribute. This could represent the name, object, cupboard and

actions to perform with them, although the implementation could choose to prompt the user for some of this information. The controls are set up to allow an entity representing the evaluator to downgrade the worker to the level of the cupboard, ie unclassified, so that it may modify it.

create_worker_functionality

Functional_Integrity
window : WINDOW
object : REF
name : NAME
identify_user_cupboard
worker : E

$object \in s.Ref[\![DRAFT \cup DOCUMENT]\!]$
$name \notin \{ i : Func[\![\{cupboard\}]\!] \cap ITEM \bullet fst(item(i)) \}$
$worker \notin entities\ s$
$s'.Class = s.Class \cup \{ worker \mapsto s.Class(window) \}$
$s'.Trust = s.Trust$
$s'.Role = s.Role$
$s'.Id = s.Id$
$s'.Conflict = s.Conflict \cup \{ worker \mapsto the\_evaluator \}$
$\exists new\_ref : REF \bullet s'.Ref = s.Ref \cup \{ worker \mapsto new\_ref \}$
$Appl' = Appl$
$changed\_flows = \{ worker \}$

create_worker

r? : R
create_worker_functionality

$r?.requestors = \{ window \}$
$r?.observed = \{ window, cupboard \}$

Downgrade Worker
The justification to allow the evaluator to perform this downgrade is not trivial, as the evaluator needs to be satisfied that the name and object chosen are not encoding any classified information. In other words a human user must be in control of the operation, and not be influenced by untrusted software in the choice of name and object to store in the cupboard. Note that if the high water mark of the requesting window is unclassified the justification is easier, as the window has accessed no classified information to encode. However, this is not expected to be the most general case. As well as becoming unclassified the worker entity is given the *dont_signal TRUST* attribute so that it may destroy itself once the cupboard has been modified.

downgrade_worker_functionality

$Trusted\_Func$
$evaluator\_entity, worker : E$

$s'.Class = s.Class \oplus \{ worker \mapsto unclassified \}$
$s'.Trust = s.Trust \cup \{ worker \mapsto dont\_signal \}$
$s'.Role = s.Role$
$s'.Id = s.Id$
$s'.Conflict = s.Conflict$
$s'.Ref = s.Ref$
$Func' = Func$

downgrade_worker

$r? : R$
downgrade_worker_functionality

$r?.requestors = \{ evaluator\_entity \}$
$r?.observed = \{ worker \}$

Worker does the Work

Finally, once downgraded the worker entity updates the cupboard, and destroys itself, ie is removed from all the functions and relations that comprise the state.

store_in_cupboard_functionality

$Trusted\_Func$
$worker : E$
$cupboard : CUPBOARD$
$object : REF$
$name : NAME$

$\exists new\_item : ITEM \bullet item(new\_item) = (name, object)$
$Func' = \{ worker \} \lhd Func \cup \{ cupboard \mapsto new\_item \}$
$s'.Class = \{ worker \} \lhd s.Class$
$s'.Trust = \{ worker \} \lhd s.Trust$
$s'.Role = \{ worker \} \lhd s.Role$
$s'.Id = \{ worker \} \lhd s.Id$
$s'.Conflict = \{ worker \} \lhd s.Conflict$
$s'.Ref = \{ worker \} \lhd s.Ref$

store_in_cupboard

$r? : R$
store_in_cupboard_functionality

$r?.requestors = \{ worker \}$
$r?.observed = \{ worker, cupboard \}$

Thus the secure operation is the sequence of secure transitions above.

$$StoreInCupboard \triangleq (\ create\_worker_{[r1?/r?]} \wedge Security_{[r1?/r?]}\ ) \fatsemi (\ downgrade\_worker_{[r2?/r?]} \wedge Security_{[r2?/r?]}\ ) \fatsemi (\ store\_in\_cupboard_{[r3?/r?]} \wedge Security_{[r3?/r?]}\ )$$

The *no_signalling* axiom requires that this sequence be initiated from the Trusted Path. In addition, the justification for the downgrade requires the human evaluator to be convinced that the human user cannot be influenced by untrusted software, in the object to store in the cupboard and the name given to it, and have information signalled through them.

StoreInCupboard keeps StoreInCupboard

4.11. Finding in the Cupboard

policy_model :Module	*Functionality :Module*	*useful :Module*
Defines *REF, R, Security*	Defines *Trusted_Func, WINDOW, NAME, ITEM, add, item*	Defines *Useful_Functions, ΞCONTROLS, identify_user_cupboard*

A window requests to be given the reference stored under a given name from the cupboard. The cupboard is discovered from the *USER* attribute of the window, as in section 4.1, and its attributes examined to find the desired item. The reference is then added to the text of the window, ie its *TEXT* attribute replaced with one representing its initial contents plus the reference, as specified in section 3.4.

from_cupboard_functionality

$$
\begin{array}{l}
Trusted\_Func \\
Useful\_Functions \\
\Xi CONTROLS \\
window : WINDOW \\
name : NAME \\
identify\_user\_cupboard \\
\hline
\exists\, i : ITEM;\ ref : REF;\ new\_text : add[\![\{(Text(window), \{ref\})\}]\!] \bullet \\
\quad i \in Func[\![\{cupboard\}]\!] \\
\quad item(i) = (name, ref) \\
\quad Func' = Func \setminus \{ window \mapsto Text(window) \} \cup \{ window \mapsto new\_text \}
\end{array}
$$

The transition is requested by the window, and the cupboard and window are the only observed entities.

from_cupboard

$$
\begin{array}{l}
r? : R \\
from\_cupboard\_functionality \\
\hline
r?.requestors = \{ window \} \\
r?.observed = \{ window, cupboard \}
\end{array}
$$

The secure operation is the combination of the functionality above and the security axioms.

$FromCupboard \triangleq from\_cupboard \wedge Security$

There is no necessity for the window to have any *TRUST* attributes as no entities are created or destroyed, and no controls are altered by this transition.

FromCupboard keeps FromCupboard

5. Discussion

This section discusses some points arising from the specifications given in the preceeding sections, and some of the less 'interesting', but otherwise useful operations that have been omitted. It concludes with a summary of the modelling exercise.

The first point to note is that although a number of secure transitions have been specified, there has not been a specification of an initial state. The reason for this is that the initial state is not interesting from the point of view of modelling the operation requirements for SERCUS, except to the extent that it must be possible for a secure initial state to exist. There are several possibilities, and for a 'real' system the initial state would probably only contain the ability for a single valid user to login and create further users, etc. The demonstration of SERCUS uses a 'compiler' to create the desired legal users, their passwords, clearances and roles, cupboards, etc. Documents, drafts and messages can also be created, using variants of the operations that work when the system is running, and placed in this 'compiled' initial state.

Note that the requirement that all the users have a unique identity and do not share cupboards, etc, can be ensured by performing the appropriate check in the transition to add a new USER attribute to the LOGIN entity. It would also be necessary to ensure that no transitions are specified which alter this information inappropriately. Thus, although an operation to allow a user to change their own password, or an operation to alter the user's role (eg promotion) could be a requirement, it would be undesirable to specify an operation which, for example, replaced the user's cupboard. However, it must be remembered that these types of requirement do not effect the confidentiality of information, but are rather requirements for functional integrity.

Mail messages and the associated operations of 'send' and 'open' have not been included in this report. This is because they have very similar properties to the cupboards and CDR, ie the users each have a mailbox and there is a central unclassified list of user identities and their mailboxes. Messages would be modelled as entities with contents attributes representing the text and header information. Opening the message would simply be a matter of the particular user observing the contents of a message entity in their mailbox, and is essentially the same as opening a document. There would also be functional integrity constraints much the same as those for draft documents to ensure that only the recipient could observe the contents. Sending a message would require the user to be on the Trusted Path to ensure that the creation of the message entity and the 'downgrade' to allow higher users to cause messages to be placed in the mailbox of lower users are not signalling. This is similar to the downgrade to allow the CDR to be updated when a new document is created.

The useful transitions which involve an entity observing and modifying itself have not been specified, for example where the *TEXT* attribute of a window is altered as information is typed in. Also the transitions where an entity observes something lower and updates its own attributes have not been included, eg where a window takes a document reference from the CDR. These have been omitted because they trivially obey the security axioms. Note that in order for the high water mark mechanism to adequately protect the information typed in by users, it should not be classified higher than the high water mark. However, this can only be enforced as a procedural control.

The open document transition gives an example of the way that error cases, such as attempts to cause information to flow against the policy, can be captured and journalled. However, although no other sort of error messages have been specified, an implementation could display error messages to the human user so long as they did not cause the software controlling the windows to be given extra information, ie these messages are not implemented as attributes of the window entities. Similarly, although logout was specified as basically a search for all the windows belonging to the user in question, this could be implemented as having a list associated with the display. This list would not be available at the abstract level directly, ie not an attribute, as it would simply be a convenient implementation detail.

An important requirement in SERCUS is that users are able to cut and paste information between the windows of their display and that the information included in any particular window is monitored using high water marks. This requirement can be modelled as an unobserved window requesting that attributes be moved between two further windows. The *no_flows_down* axiom is trivially satisfied as all the windows will be classified at the clearance of the user. The *Trusted_Func* axiom will ensure that the high water mark of the modified window (ie paste) is raised to reflect the high water mark of the

observed window (ie cut). Because the requestor of this transition is modelled as unobserved, this requires that the human user requested the cut and paste operation rather than any untrusted software acting on their behalf. Where the implementation functionality desires that arbitrary data structures be cut and pasted between windows, for example pictures which cannot be attributes unless they can be guaranteed to be immutable, this can be modelled as the joining of the two window entities with a single high water mark. Again this must be requested by the human user.

Note that active software in SERCUS is generally modelled by window entities, but that this does not necessarily require a visible representation on the users display. Another active entity in the operation specifications is the worker entity used when the functionality requirement is split into a sequence of transitions. This could be considered as modelling a procedure call. In other words the window calls a procedure which carries out the sequence of transitions and then exits, returning control to the window. This worker is only given the attributes required to carry out its tasks (parameters), and is generally used when a regrade is required. It would be inappropriate to regrade the original requesting window, even if it was returned to its original level after the sequence of transitions, as this could lead to either its attributes being underprotected or it becoming cleared to see inappropriate information for the duration of the sequence of transitions. It is important to remember that the model deals with secure transitions and does not impose any ordering on the individually secure transitions. The separation of duty controls on the worker entity imply that, in an implementation, the code is trusted not to abuse its privileges and that its workspace is protected from observation, ie not implemented as an attribute. This is captured in the model by the use of the evaluator role to justify the clearance and trust given to the worker.

As discussed in section 3.11, where a requirement has to be modelled as a sequence of individually secure transitions, a desire for functional integrity requires that information be passed between them to ensure that the later transitions act upon the results of the previous ones. The properties of schema composition ensures that the attributes and entities identified in the signatures with the same name, are the same value. Consequently, the Z specification does specify that later transitions act upon the desired entities. However, providing functional integrity in this way implicitly requires that the composition of transitions is trusted. In other words, not only would this require that the code implementing the individual transition be correct, but that they could be shown to be called in the correct order and with the correct parameters. However, as these flows of parameters between transitions would not be explicitly modelled in terms of entities being modified with attributes, the model itself could provide no guidance to the implementation, such as which flows are downward.

However, neither is it appropriate to modify entities explicitly with all these parameter attributes, as in many cases the actual information required depends on the implementation's representation of the abstract attributes and the decision as to whether to prompt the human user for input. For example, when a reference is named in the cupboard, the implementation could either choose to observe all the information from the requesting window, or prompt the human user for the name to use. Another example would be where the text of a window has the reference to a newly created document added to it. At this abstract level the only property specified about text is the references it contains, however, the implementation could also wish to include the literal representation of the new CDR number.

Therefore, this specification provides for functional integrity by the modification of entities which could require parameters from previous transitions in the sequence with *FLOW* attributes. These are abstract attributes representing whatever information the implementation requires, although it must be stressed that this can only come from the nominated observed entities and unclassified control information. In addition, in order to preserve their 'attributeness', ie immutability, any changes to the information that a *FLOW* attribute is representing must result in a new attribute at the abstract level. Therefore the implementation of *FLOW* attributes and sequencing transitions to provide the desired functionality cannot result in undesirable flows of information.

It has been noted already, in sections 3.9 and 4.2, that the classification of the *LOGIN* entity does not protect its attributes, ie user identities and passwords, from observation. It is useful to stress again the fact that this information does not have a classifiaction associated with it in the same way as, for example, documents do. The actual requirement is simply that this information is not observed or modified 'inappropriately'. Thus, in terms of the Terry-Wiseman Model, which is solely concerned with the flows of classified information, the protection of information without a classification requires that the system owners are satisfied that there are no inappropriate transitions, and could also involve encrypting the data. Thus the adequate protection of passwords can be viewed as a 'proof opportunity'.

Another example where the classification controls are too coarse is draft documents. Here the drafts are classified at the clearance of the user since they are able to type information into the system up to this level. In addition, the users can move information from windows into drafts, etc, and their windows are classified at their clearance. However, the actual functionality requirement is that drafts are private to the user until they have been given a final classification and journal and have been placed in the central registry. Thus, although the confidentiality controls would permit any user sufficiently cleared to view anothers draft, SERCUS has to ensure that this does not happen. This specification has chosen to model this additional access control using the human identities that already existed for the separation of duty controls. The operations concerning draft documents then simply check that the *ID* control attribute from the draft and requestor are the same. Since each draft only ever belonged to a single user this seemed to be an appropriate way to model this identity based access control. The modelling of more general access control methods is discussed in [Harrold90a].

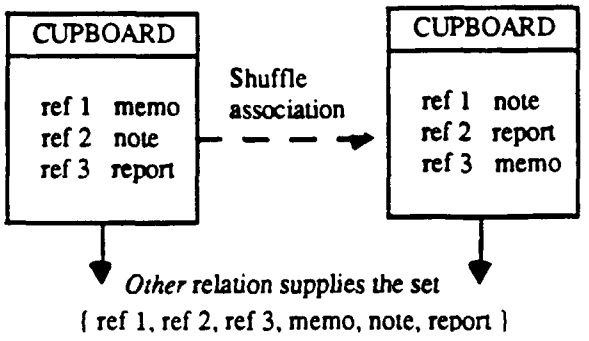
An implementation providing, for example, virtual machines or a hierarchical filing system, would have no difficulty keeping the existence of another's draft objects hidden, and the id information and additional check would not be required. In fact, the model's view that the existence of all entities is freely visible and insistence that this is unclassified can lead to it being overstrong, particularly in the area of secure databases. Work is in hand to produce a hierarchical version of the model which would help with these considerations.

Associated with the problems of this additional access control on draft entities, is the potential signalling channel through its high water mark. The software editing the draft can manipulate the level of the high water mark on the basis of classified information, ie by choosing whether or not to access an object that would cause the level to float higher, up to the clearance of the user. In a classification system with a large number of caveats and categories there are many possibilities. This is not a problem until a new document is created from the draft, as the high water mark is protected at the maximum level of the encoded information (ie clearance of the user). The classification of the document reveals information about the high water mark, either its exact value if the users always classify documents at the level of the draft's high water mark, or at least that it was no higher than the document classification. However, as this channel is only one bit per creation of a document and can only occur in SERCUS when the human user on the Trusted Path creates a document, and since the untrusted software is 'memoryless' between invocations, this is not considered to be a threat in SERCUS.

It has already been noted that where a requestor is unobserved this implies that the human user initiated the transition using a Trusted Path and supplied the parameters. It is also important to note the implications of a transition where an entity is modified without being also observed, for example the update of the highly classified journal. In such cases the journal must be unobserved in all circumstances. In other words the update must never fail. Thus, the requestor believes the journal has been modified even if, for example, it was already full. The implications of the requirement to audit, ie observe, this high information were discussed in section 3.5. Finally, it is useful to note again the implications of transitions where the requestors are unmodified, for example the regrade document operation discussed in section 4.9. Here the requestors do not know whether all parties agreed to the transition and it did in fact take place. Consequently the requestors need not all be classified at the same level as they are unable to signal their information to each other through their decision as to whether the transition may take place or not.

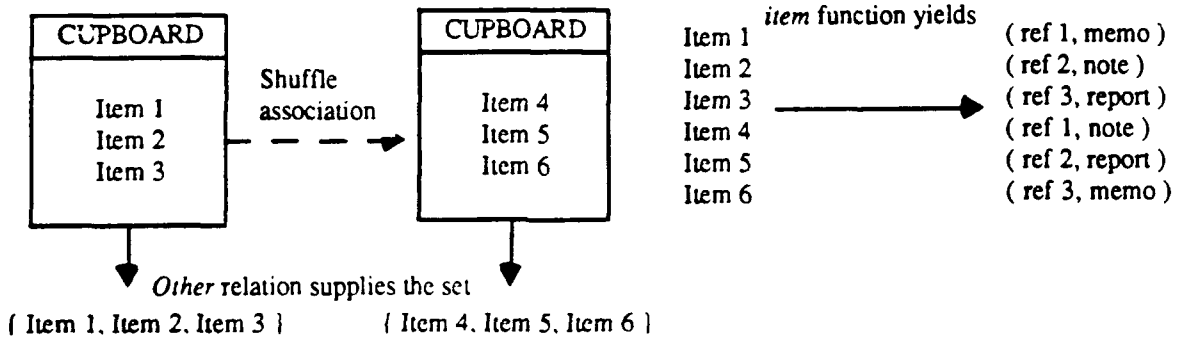
Section 3 provided several examples of the use of functions and relations to provide structure to attributes, eg the *ITEM* attributes of cupboards and the function *item* to associate a name and reference pair to each (section 3.6). This structuring is necessary because the model's view of functionality is deliberately very abstract in order that its confidentiality controls may be generally applicable. Thus the *Other* relation simply identifies the set of attributes contained in each entity and applies the security axioms whenever attributes are added or removed from this set. Therefore should the items in a cupboard be modelled by name and reference attributes. as in Figure 8a below, mixing the names up would not be captured by the model's definition of a transition, and the security axioms would not be applied.

Figure 8a: Unstructured Attributes



This is not a problem with the model but with the level of abstraction being used to model the functionality. If there are important relationships between the various functionality aspects of an entity, they must be modelled, and the simplest way is to model the relationships as attributes and provide a means to recover the aspects of interest, as illustrated by Figure 8b.

Figure 8b: Structured Attributes



Note that in the case of the events added to a journal (section 3.5) no functions are defined as the actual information contained in an event is not of interest at this level of abstraction. Similarily with *TEXT* attributes, only the function to supply the set of references contained in it is defined. as the ordering of these references, the words or characters, the position of the cursor, etc, are not of interest at the level of abstraction of this specification.

It is important to remember that the separation of duty controls do not always require the physical presence of the 'n' requestors. In many cases the agreement of at least one of the parties is automated by the use of trusted software, for example certain downgrades are justified using high water marks. This is discussed further in [Harrold90b]. Also, although the separation of duty controls are modelled as happening in parallel at the abstract level, they are implemented as a sequence. However, it is important that any sequencing information does not introduce flows of information that were not considered at the higher level. For example, as the regrade to a document is specified by the simultaneous agreement of the user and security officer, the implementation will introduce extra structure to record the request until the security officer authorises it. Should this request to regrade be observable by further entities this introduces 'modification' to the document and a flow of information to this entity which was not specified nor considered at the abstract level.

There are several examples where the justification for a downgrade has been specified by the use of an evaluator role in the separation of duty, eg the downgrade to the worker entity to allow the unclassified CDR to be updated when a new document is created (section 4.7). This has proved necessary in the cases where the model itself cannot supply a justification because this rests on factors outside the scope of the model. Examples of this include the cases where the implementation code is trusted not to mix up variables of differing classifications; the downgrade is not considered to be a threat from the result of a risk analysis (human judgement); or that certain procedural controls have been enforced. However, this evaluator entity is not treated in quite the same way as other active entities as it is neither observed nor

modified by any transitions. The human evaluators of the system are not required every time, for example, a document is created. They evaluate and approve the code instead, taking into account many factors outside the scope of the model (eg development environment, etc), and will agree that the update to the CDR as part of the creation of a document is an acceptable 'downward flow'. Thus, the implementation code becomes 'trusted' and the evaluators are no longer directly involved. They provide no information for the creation of each document and neither do they know any details of the documents created. Therefore, the evaluator role could simply be viewed as indicating the places where code needs to be 'trusted' to provide the desired functionality.

Finally, although each operation has been considered in relation to the five axioms defining the security requirements, this has been informal. The opportunity exists for formally proving that the precondition of each operation is true, ie the functionality is not contradictory to the security.

The model has been unforgiving in its insistence that a justification be given for each flow of information that it considers to be 'downward'. In many cases these downward flows were 'obviously secure' and the justification was trivial. However in other cases it provoked a lot of thought and the realisation that the requirement was vulnerable to certain kinds of Trojan Horse attack. This is a major advantage of the model, namely the fact that the controls are modelled along with the information they protect and the interactions between them can be reasoned about. Another advantage is the fact that the particular aspects of trust that are required are identified.

A significant amount of effort has gone into modelling the SERCUS requirements. Much of this was in exploring the advantages and disadvantages of various approaches, and further papers are in preparation which detail these arguments. Thus, although the specification given in this report is not the only method, experience with other approaches has led to the conclusion that it is the most appropriate.

It must also be noted that modelling SERCUS has been an iterative process where certain classes of operation were considered, often resulting in an improvement to the model or a clarification to its interpretation. Thus the original modelling work began with the model presented in [Terry&Wiseman89] and has concluded with the model presented in Annex B and in [Harrold90a]. Section 6 of [Harrold90a] details this evolution.

In summary, this report has illustrated the way in which the Terry-Wiseman Model can be used to model the requirements of a non-toy application which requires that the confidentiality of its information is assured.

Acknowlegements
Finally I would like to acknowledge the contributions of Simon Wiseman and Phil Terry, and also to thank Peter Bottomley, Sharon Lewis and Andrew Wood for their comments on earlier drafts of this report.

6. References

[Bottomley& Wiseman88] SMITE - RSRE's Computer Architecture for Multi-level Security
P C Bottomley, S R Wiseman
proceedings of Milcomp 1988, pages 25 - 30

[Harrold88] Formal Specification of a Secure Document Control System for SMITE
C L Harrold
RSRE Report 88002, February 1988

[Harrold89] An Introduction to the SMITE Approach to Secure Computing
C L Harrold
Computers and Security Journal, October 1989, 8 (1989) 495-505

[Harrold90a] The Terry-Wiseman Security Policy Model and Examples of Its Use
C L Harrold
RSRE Report 90001, March 1990

[Harrold90b] A Discussion about the Role of Separation of Duty in Maintaining Security
C L Harrold
to be published

[King*etal*87] Z: Grammar and Concrete and Abstract Syntaxes
S King, I H Sorensen, J Woodcock
Programming Research Group, Oxford University, July 1987

[Randell90] Zadok User Guide
G P Randell
RSRE Memorandum 4356, January 1990

[Sennett87] Review of the Type Checking and Scope Rules for the Specification Language Z
C T Sennett
RSRE Report 87017, November 1987

[Spivey88] The Z notation: A Reference Manual
J M Spivey
Prentice-Hall International 1988. ISBM 0-13-983768-X

[Terry89] The SMITE Approach to Security
P F Terry
RSRE Report 89014, August 1989

[Terry&Wiseman89] A 'New' Security Policy Model
P F Terry, S R Wiseman
proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA
May 1989, pages 215 - 228

[Wiseman*etal*88] The Trusted Path between SMITE and the User
S R Wiseman, P F Terry, A W Wood, C L Harrold
proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA
April 1988, pages 147 - 155

[Wood88] A Z Specification of the MaCHO Interface Editor
A W Wood
RSRE Memorandum 4247, November 1988

[Woodward87] Exploiting the Dual Nature of Sensitivity Labels
J P L Woodward
proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA
April 1987, pages 23 - 30

Annex A: An Overview of the Z Notation

Z is a powerful mathematical notation that has been developed by the Programming Research Group at Oxford University. The underlying basis of Z is standard set theory, and it makes use of the associated notation. Properties about sets are described using predicate calculus. A Z specification is structured into self contained parts using schemas.

New sets are introduced between square brackets, for example, [*BOOK*] introduces the set of all possible books, or using == followed by a definition of the set.

$\{\}$	empty set
$\in$	LHS is a member of the set on the RHS
$\notin$	LHS is not a member of the set on the RHS
$\subseteq$	set on the LHS is a subset of the one on the right (possibly equal)
$\subset$	set on the LHS is a proper subset of the one on the right (never equal)
$\cup$	result is the union of the two sets
$\cap$	result is the intersection of the two sets
$\bigcup$	distributed union of a set of sets, result is the union of all sets
$\setminus$	result is the set equal to the LHS with members of the RHS removed
$\neq$	inequality
#	number of elements in a set
$\mathbb{N}_1$	the set of strictly positive natural numbers
$x : T$	declaration, x is an element drawn from the set T
$\{ D \mid P \bullet t \}$	the set of t's from the declarations D such that the predicate P holds
$\mathbb{P}$	powerset, ie the set of all possible subsets of a particular set

Thus, *childrens* : $\mathbb{P}$ *BOOK*, says that the set of books for children is 'drawn from the set of subsets' of the set BOOK, ie childrens books are a subset of all books.

A relation may be viewed as a set of ordered pairs. Functions are a special type of relation where there is a single element in the range for each element of the domain. Thus the operators defined for sets are applicable to both functions and relations.

dom	domain of a relation, ie all the first elements of the ordered pairs
rng	range of a relation, ie all the second elements of the ordered pairs
fst	first element of an ordered pair
$\leftrightarrow$	relation
$\rightarrow$	total function, domain is all possible members of the set
$\nrightarrow$	partial function
$\rightarrowtail$	injective partial function, each element in range associated with only one in domain
$\rightarrowtail$	injective total function
$\mapsto$	maplet, graphic way of expressing an ordered pair
$[\![ ]\!]$	relational image, $R[\![S]\!]$ the set related by the relation R to the members of the set S
R^{-1}	inverse of the relation R
$R \rhd S$	range restriction, restrict relation R to those range elements in set S
$R \rhd S$	range substraction, take out of relation R those pairs with range element in set S
$S \lhd R$	domain restriction, restrict relation R to those domain elements in set S
$S \lhd R$	domain substraction, take out of relation R those pairs with domain element in set S
$S \fatsemi R$	relational composition, ie S followed by R (type of rng S must equal type of dom R)
$f \oplus g$	functional override, replace relevant pairs in function f with g. (dom g $\lhd f$) $\cup$ g

;	list separator
$\wedge, \neg$	and, not
$\Rightarrow, \Leftrightarrow$	implication, equivalence
$\forall x : T \bullet P$	for all x of type T predicate P holds
$\exists x : T \mid D \bullet P$	there exists an x of type T for which predicates D and P hold
$\exists_1 x : T \mid D \bullet P$	there exists a unique x, ie one and only one x satisfying D and P
Predicates	Where clause, shorthand for $\exists D \bullet P$. Declarations only in scope until
where	end of predicates.
Declarations	

declaration	an axiomatic definition, the declarations are global the predicates define properties about them
predicates	

name *signature*	a schema, the signature declares some variables and their types the predicates define properties about them
predicates	the objects declared in one schema are made available to another by including the name of the schema in the signature. They are then in scope until the end of this schema. Schemas can also be used as types.

Axiomatic definitions and schema names are in scope from their point of introduction until the end of the Z module.

The following conventions are used for variable names in schemas representing operations:

undashed basename	state before
dashed, ie ending in '	state after
ending in ?	inputs or parameters
ending in !	outputs or results

$s.t$	project the variable t from the schema variable s
$\widehat{=}$	schema definition
$S \wedge T$	S and T, resulting schema is formed by merging the declarations from S and T and by conjoining their predicates.
$S \fatsemi T$	schema composition, where the basenames are the same the state after components in S become the initial state components in T. All other components of the schemas are anded together.
$S \oplus T$	schema overide, schema S unless T is applicable (ie its precondition is true), $(S \wedge \neg\, pre\ T) \vee T$

Annex B: The Formal Model and Its Interpretation

This annex contains the Z specification of the Terry-Wiseman Model that is used throughout this paper, and is essentially section 3 RSRE report 90001 "The Terry-Wiseman Security Policy Model and Examples of Its Use".

Set Definitions

First the sets of all possible entities and attributes are introduced.

$[E, A]$

Next, a subset of all possible attributes is identified as representing classifications. It is also necessary to bring in the notions of the dominance relationship, >=, between classifications, and the least upper bound, *LUB*, and greatest lower bound, *GLB*, operators on sets of classifications. As these concepts are well understood, the actual definitions have been omitted. Thus, the classification of an entity will be represented by an attribute drawn from the set called CLASS. Note that in some cases this may be interpreted as a clearance.

$CLASS : \mathbb{P}\, A$

$\_>=\_ : CLASS \leftrightarrow CLASS$
$GLB,\ LUB : \mathbb{P}\, CLASS \rightarrow CLASS$

A further subset of attributes are identified as representing the various types of TRUST that can be given to active entities. Three particular types of TRUST attribute are identified.

The *dont_signal* attribute will be given to those entities which are assumed not to exploit signalling channels, or otherwise to write classified information into the controls. The basis of this assumption may be that a human is in control, or that some other controls are being deployed to close the channel.

The *faithful* attribute will be given to entities which always do the bidding of a human user, in other words the human user is accountable for the actions of these entities, whether directly through a trusted path, or indirectly because the software will always make the same decision/actions as the human. Faithful entities must be both functionally correct and free from Trojan Horses, ie proven to meet their specification.

Finally, the *creator* attribute will be given to those entities which are trusted to correctly set up the security controls on any new entities. The basis of this trust is application specific.

$TRUST : \mathbb{P}\, A$
$dont\_signal,\ faithful,\ creator : TRUST$

The set of all possible roles and the unique identifiers of the human users of the system are identified as further subsets of all possible attributes. No specific roles are identified here as these are application specific. However, particular examples could be security officer, librarian, bank manager, counter clerk or system owner.

$ROLE : \mathbb{P}\, A$
$ID : \mathbb{P}\, A$

A further subset of attributes is identified to represent all possible conflicts, ie the n man rules, for the separation of duty controls. A function is defined, *conflict_roles*, which gives the numbers of each type of role required. For example, the conflict attribute of a document could require one security officer and one ordinary user to agree to any alteration in its controls. All CONFLICT attributes uniquely have such numbers and roles associated with them.

$CONFLICT : \mathbb{P}\, A$

$$conflict\_roles : CONFLICT \rightarrowtail (ROLE \nrightarrow \mathbb{N}_1)$$

$$\{\} \notin rng\ conflict\_roles$$

Finally the set of attributes which represent references to entities, ie the means of addressing entities, is identified.

$$REF : \mathbb{P}\ A$$

The State

The state of the machine is given by the following schema, which structures the state into a number of named functions and relations between entities and attributes.

STATE

$$Class : E \nrightarrow CLASS$$
$$Trust : E \leftrightarrow TRUST$$
$$Role : E \leftrightarrow ROLE$$
$$Id : E \nrightarrow ID$$
$$Conflict : E \leftrightarrow CONFLICT$$
$$Ref : E \twoheadrightarrow REF$$
$$Other : E \leftrightarrow A$$

Class is the partial function which supplies the classification attribute of an entity. It is a function (many-to-one) as entities may only have a single classification and partial as nothing is specified about those entities which do not exist in the current state, ie those yet to be created or which have been destroyed.

Trust is the relation which supplies the various types of trust invested in an active entity. It is a relation (many-to-many) as entities may have more than one trust attribute, and several entities may be similarily trusted.

Role is the relation which supplies the particular role (or roles) that an active entity plays in the system.

Id is the function which supplies the unique identity of the human user that an active entity is representing. It is a function as entities may represent at most one human, and partial as not all entities are active and so that entites may be both created and destroyed.

Conflict is the relation which identifies the various numbers and types of roles that must cooperate in any alteration to the security controls of an entity. It is a relation so that more than one conflicting set may be specified. Note that a single role is permissable even though it does not provide for any conflict of interest. This allows for an application to have very privileged and trusted entities, for example the system owners themselves. Also, if the requirement is that the controls of an entity are unalterable this should not be specified as no roles but as the set of all possible roles. This is because an empty set could allow unfaithful entities acting on their own to alter the controls.

There is one conflict set which covers modifications to all the security controls. However, there is no reason why a slightly different model could not be specified, with different conflicts for different controls, if this was the functionality required.

Ref supplies the means to address an entity, ie its reference. This is a function as each entity has a single reference, and partial to allow entities to be created and destroyed. The function is injective (one-to-one) as references may only be associated with a single entity, thus making references unique.

Other encompasses all the functionality aspects of the system, and is simply a relation between

entitites and attributes. Further, it is the attributes in the range of this relation which are considered to be protected by the controls. Therefore, this model is of all possible applications where confidentiality is the prime concern.

Note that the control type attributes may be found in the functionality attributes, for example, entities may contain references to other entities. Further, the various sets of attributes are not necessarily disjoint, and therefore a particular attribute may actually represent different things depending on context. For example, should integers be used to represent both classifications and months of the year, then the number 1 may mean Top Secret when it is representing the classification of an entity or January when part of the functionality.

Supporting Definitions

In defining the security policy model a number of functions are needed to characterise transitions in terms of the differences between states.

The following schema defines the symmetric set difference operator, $\uparrow$. This supplies all the differences between two sets. A second operator, $\downarrow$, is defined. This turns a relation into a function giving a set.

$$
\begin{array}{|l}
[X,Y] \\
\hline
(\_\uparrow\_) : (\mathbb{P} X \times \mathbb{P} X) \rightarrow \mathbb{P} X \\
\downarrow : (X \leftrightarrow Y) \rightarrow (X \nrightarrow \mathbb{P} Y) \\
\hline
\forall x, y : \mathbb{P} X \bullet x \uparrow y = (x \cup y) \setminus (x \cap y) \\
\forall r : X \leftrightarrow Y; x : dom\ r \bullet dom (\downarrow r) = dom\ r \\
\qquad\qquad \downarrow r(x) = r(\!| \{x\} |\!) \\
\hline
\end{array}
$$

The following schema defines an operator, *flatten*, which returns all the entity-attribute relationships that comprise a state. All the entities that exist in a state may then be discovered, using the operator *entities*, which applies the domain operation to a flattened state.

$$
\begin{array}{|l}
\hline
flatten : STATE \rightarrow (E \leftrightarrow A) \\
entities : STATE \rightarrow \mathbb{P} E \\
\hline
\forall s : STATE \bullet flatten\ s = s.Class \cup s.Trust \cup s.Role \cup s.Id \cup s.Conflict \cup s.Ref \cup s.Other \\
\qquad\qquad entities\ s = dom\ (flatten\ s) \\
\hline
\end{array}
$$

State Transitions

State transitions are considered to be requested by a set of entities, called the *requestors*. This is a set, rather than a single entity, in order to capture the notion of separation of duty for integrity. The second set of entities identified in the request are those entities which were *observed* during the transition. An entity is observed if its contents (ie the attributes defined by the Other relation) had any influence over the outcome of the state transition. It is very important that in an implementation entities not identified as observed have no influence over the outcome of a state transition. This observed set is identified in the transition request because although it is possible to examine the state to identify the receivers of information flow, ie those entities which had attributes changed, gained or lost, it is impossible to identify the source of the flow in the same way.

$$
\begin{array}{|l}
R \\
\hline
requestors, observed : \mathbb{P} E \\
\hline
\end{array}
$$

Note that the requestors of a state transition need not be observed. For example a command line interpreter decides what action to take on the basis of human input rather than its state. Also, the requestors need not be modified by the transition. Each requestor may say yes or no to the change as appropriate but will not necessarily remember whether all concerned agreed and the transition took place or not. For example, security officers may not remember whether they authorised a particular downgrade, although they may be able to look up the fact in a journal at a later date.

Valid state transitions are given by the following schema, which identifies the request and the before and after states. There are two constraints put on a valid state transition. The first is that transitions only occur if they are requested by entities that exist in the current state, and secondly, that the observed entities also exist.

TRANSITION

$$r? : R$$
$$s, s' : STATE$$

$$\{\} \subset r?.requestors \subseteq entities\ s$$
$$r?.observed \subseteq entities\ s$$

The Security Axioms

This section formally defines security axioms to provide confidentiality and integrity.

Confidentiality is divided into two separate concerns. The most obvious aspect of confidentiality is that information is not moved from highly classified to lowly classified containers. Relating this to the world of people looking at documents, for example, the windows of users' displays, which will be labelled with their clearance, cannot gain attributes from a document classified higher than that clearance.

The following schema, *no_flows_down*, identifies the set of entities whose view of the state was in some way modified. This view excludes the control attributes, as other axioms ensure that the controls on an entity are not classified. It also excludes entities which were destroyed, but does include newly created entities. These *modified* entities are therefore the receivers of the information flow. The source of the flow is simply those entities whose contents were observed.

Thus the axiom simply states that after the transition the lowest classification of the modified entities must dominate the highest classification of observed information before the transition. Thus information cannot flow down into either existing or newly created entities. Note that information may flow unconstrained between entities without classifications, and hence it is necessary to ensure that all entities have a classification, see the *Correctness* aspect of integrity below.

no_flows_down

TRANSITION

$$GLB\ s'.Class[\![modified]\!] >= LUB\ s.Class[\![r?.observed]\!]$$

where

$$modified == dom\ (\ s.Other \uparrow s'.Other\ ) \cap entities\ s'$$

The second aspect of confidentiality is that signalling channels are not exploited by untrusted software, formally expressed by the *no_signalling* axiom below. The signalling channels identified as *changed_controls* capture the channels through changing any of the security controls on existing entites and also through the creation and deletion of entities. The non-exploitation is expressed by insisting that should any of these aspects of the state be altered by a transition, then all the requestors must possess the *dont_signal* trust attribute.

no_signalling

TRANSITION

$$changed\_controls \neq \{\} \Rightarrow (\forall r : r?.requestors \bullet dont\_signal \in s.Trust(\!| \{r\} |\!))$$

where

$$\begin{aligned} changed\_controls == \quad & dom(s.Class \uparrow s'.Class) \\ & \cup dom(s.Trust \uparrow s'.Trust) \\ & \cup dom(s.Role \uparrow s'.Role) \\ & \cup dom(s.Id \uparrow s'.Id) \\ & \cup dom(s.Conflict \uparrow s'.Conflict) \\ & \cup dom(s.Ref \uparrow s'.Ref) \\ & \cup (entities\ s \uparrow entities\ s') \end{aligned}$$

Thus, the confidentiality of information can be expressed as the conjuction of the above two aspects.

$Confidentiality \triangleq no\_flows\_down \wedge no\_signalling$

In this model of security the prime concern is the confidentiality of information. Consequently the integrity of the controls that enforce confidentiality is also of concern. However it must be stressed that the integrity of the protected information is an application specific issue. Integrity is divided into two aspects. Firstly, entities must have the necessary control attributes in order that the confidentiality controls may be applied, and secondly these controls must be in some way appropriate to the information they protect.

The *Correctness* schema simply insists that in order for any transition to occur all the entities involved in the transition must have a classification attribute. Thus the confidentiality controls may be applied. These entities must also have a reference attribute in order to be addressable by an implementation. Nothing further is said about the particular addressing mechanism.

Note that not all entities need have trust, role or id attributes. These are only required by the active entities, ie those requesting transitions, and only then if they request security critical transitions. The particular separation of duty controls placed on entities is application specific. However, note that the set of conflicting roles for separation of duty should be non-empty, otherwise, unfaithful entities acting on their own could potentially alter the controls. Thus the following schema insists that all the entities involved in a transition have a classification, reference and at least one conflict attribute. Similarily a transition must preserve these properties. Nothing is said about entities destroyed by a transition except that they had the necessary attributes beforehand.

Correctness

TRANSITION

$$
\begin{aligned}
&before \subseteq dom\ s.Class \quad \wedge\ after \subseteq dom\ s'.Class \\
&before \subseteq dom\ s.Ref \quad \wedge\ after \subseteq dom\ s'.Ref \\
&before \subseteq dom\ s.Conflict \ \wedge\ after \subseteq dom\ s'.Conflict \\
&\textit{where} \\
&\quad involved == r?.requestors \\
&\qquad \cup\ r?.observed \\
&\qquad \cup\ dom(\ s.Class \uparrow s'.Class\ ) \\
&\qquad \cup\ dom(\ s.Trust \uparrow s'.Trust\ ) \\
&\qquad \cup\ dom(\ s.Role \uparrow s'.Role\ ) \\
&\qquad \cup\ dom(\ s.Id \uparrow s'.Id\ ) \\
&\qquad \cup\ dom(\ s.Conflict \uparrow s'.Conflict\ ) \\
&\qquad \cup\ dom(\ s.Ref \uparrow s'.Ref\ ) \\
&\qquad \cup\ dom(\ s.Other \uparrow s'.Other\ ) \\
&\quad before == involved \cap entities\ s \\
&\quad after == involved \cap entities\ s'
\end{aligned}
$$

Thus, the first aspect of integrity basically ensures that entities have the correct attributes in order that the confidentiality controls may be applied. The second aspect of integrity is that these control attributes are in some way appropriate to the information they protect. This is divided into two parts. The first concerns itself with modifications to controls of existing entities and the second with the controls given to new entities.

Separation of duty is used as the means to ensure the appropriateness of changes to security controls of existing entities. The *Separation_of_Duty* axiom below states that whenever any controls were modified by a transition there were sufficient requestors with the necessary conflicting roles, and further that these requestors were acting on behalf of an appropriate number of different human users, ie they possessed the *faithful* trust attribute, and had different ID attributes. It is important to note that because of possible collusions amongst the roles, separation of duty does not in itself guarantee security. However, sensible use of the mechanism does provide the best control that can be realistically achieved.

Separation_of_Duty

TRANSITION

$$
\begin{aligned}
&\forall\ e : modified\_controls \bullet \\
&\quad \exists\ f : ID \nrightarrow ROLE \mid f \subseteq s.Id^{-1} \fatsemi (\ faithful\_requestors \lhd s.Role\ ) \bullet \\
&\qquad \downarrow f^{-1} \fatsemi (\#\_) \in (s.Conflict \fatsemi conflict\_roles)(\!|\ \{e\}\ |\!) \\
&\textit{where} \\
&\quad modified\_controls == entities\ s \cap entities\ s' \cap (\ dom(\ s.Class \uparrow s'.Class\ ) \\
&\qquad \cup\ dom(\ s.Trust \uparrow s'.Trust\ ) \\
&\qquad \cup\ dom(\ s.Role \uparrow s'.Role\ ) \\
&\qquad \cup\ dom(\ s.Id \uparrow s'.Id\ ) \\
&\qquad \cup\ dom(\ s.Conflict \uparrow s'.Conflict\ ) \\
&\qquad \cup\ dom(\ s.Ref \uparrow s'.Ref\ ) \\
&\qquad)\ \\
&\quad faithful\_requestors == \{\ r : r?.requestors \mid faithful \in s.Trust(\!|\ \{r\}\ |\!)\ \}
\end{aligned}
$$

The second aspect of appropriateness concerns itself with the controls given to new entities. The *Trusted_Creation* axiom below states that whenever entities are created by a transition, at least one of the requestors was trusted to correctly set up the controls, ie possessed the *creator* trust attribute. Note that this trust covers the appropriateness of the controls that are given to the entity and also ensures that no necessary controls are omitted. Also note that the classification given to new entities is constrained by the *no_flows_down* axiom which insists that it be at least as

high as the highest observed information. Establishing the basis of creator trust is application specific, and could, for example, also be enforced using separation of duty controls.

Trusted_Creation

TRANSITION

$$entities\ s' \setminus entities\ s \neq \{\} \Rightarrow creator \in s.Trust(\!|\ r?.requestors\ |\!)$$

Thus, ensurance of the appropriateness of control attributes is the combination of separation of duty controls upon any modifications and trust upon creation.

$$Appropriateness \triangleq Separation\_of\_Duty \wedge Trusted\_Creation$$

Integrity is seen to be the combination of the two aspects of correctness and appropriateness.

$$Integrity \triangleq Correctness \wedge Appropriateness$$

Finally, security is defined to be the confidentiality of information together with the supporting integrity of the controls that are used to enforce the confidentiality.

$$Security \triangleq Confidentiality \wedge Integrity$$

The following line of Z defines this annex to be a module and exports the definitions for use in other specifications.

policy_model keeps E, A, CLASS, >=, GLB, LUB, TRUST, dont_signal, faithful, creator, ROLE, ID, CONFLICT, conflict_roles, REF, STATE, $\uparrow$, $\downarrow$, flatten, entities, R, TRANSITION, no_flows_down, no_signalling, Confidentiality, Correctness, Separation_of_Duty, Trusted_Creation, Appropriateness, Integrity, Security

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) .....

Overall security classification of sheet .....UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No.	Month	Year
REPORT 90011	JULY	1990

Originators Name and Location
RSRE, St Andrews Road
Malvern, Worcs WR14 3PS

Monitoring Agency Name and Location

Title
AN EXAMPLE SECURE SYSTEM SPECIFIED USING THE TERRY-WISEMAN APPROACH

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period

Project Number	Other References

Authors	Pagination and Ref
HARROLD, C L	60

Abstract

This report presents the specification of operations for a secure document handling system (SERCUS). The specification uses the Terry-Wiseman Security Policy Model and therefore acts as an example of the modelling approach. The specification uses the mathematical notation Z, and consequently also acts as an example of the use of Z in specifying secure systems. However, it must be noted that an appreciation of SERCUS, the model and modelling approach can usefully be gained even if the formal specifications are not read. The Terry-Wiseman Model and its interpretation are given as an Annex to this report.

Abstract Classification (U,R,C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)

UNLIMITED