Chapter 3

Specification and Validation of a Network Security Policy Model

Peter Lindsay

Summary

This case study concerns the specification and validation of a Security Policy Model (SPM) for an electronic network. The network is intended to provide processing and transmission services for electronic messages, including sensitive and classified material, over distributed sites and supporting multiple levels of security classification. The SPM is formally specified in VDM-SL and validated by showing that the model is mathematically consistent and satisfies certain security properties. Rigorous proofs are provided. In addition, the case study illustrates some new techniques concerning proof obligations for exception conditions in VDM-SL.

3.1 Introduction

3.1.1 Background and Context

This chapter describes the specification and validation of a formal Security Policy Model (SPM) for an electronic-message processing and transmission service. The SPM is a distillation of the important security requirements of the software system that provides the service. The SPM described here is based on a security model originally proposed for an Australian Government agency's secure distributed network; the model has been changed in certain ways, however, to protect sensitive details.

A high degree of assurance in the correctness of the model and the system's security was required – roughly equivalent to the requirements for level E5 in the ITSEC computer security standard [1]. A particular accreditation requirement was that the SPM be described in a formal language and that formal proofs of correctness be performed. This chapter describes a formal specification of the SPM in VDM-SL [4], but rather than present formal proofs, the proofs are given rigorously here, for clarity and ease of understanding.

By way of context for the specification and validation of the SPM, the overall computer system security objectives will be outlined in the rest of this section. The process by which the security objectives were attained will not be discussed here, however, nor how particular aspects of the model were determined to be the appropriate ones for study. The interested reader is referred to Landwehr's excellent survey article [6] for explanation of computer security terminology and for further background on the use and need for security models.

3.1.2 Software System Requirements

The system's primary function is to provide a secure message processing and transmission service for a government agency whose offices are distributed across many locations. Messages generated and processed on the system range in sensitivity from unclassified to classified material with different levels of security classification. The system is also required to provide a message transmission service for other government agencies, which send and receive messages via the network.

An important feature of the agency's message processing procedures is analysis of messages. Each message is subjected to review by one or more analysts, to determine if the content of the message warrants additional dissemination and whether additional relevant information should be appended to the message. Analysis occurs both at the location where the message is generated (by a local expert) and at the organisation's headquarters by a team of experts (called central analysts).

In outline, the processing activities applied to a message from conception to delivery are as follows:

1. The author generates a message together with its classification and a list of proposed recipients.
2. The author sends the message for review by a local analyst, which may result in information being added, the destination list changing, or in the message's classification being modified.
3. The local analyst sends the message for review by central analysts, which may result in similar modifications of the message. Central analysis may involve review by one or more analysts, depending upon the message's content.
4. When central analysis is complete, the message is added to a queue for delivery.
5. The system transmits the message to appropriate locations, where local analysis may take place prior to delivery of the message to its intended recipients.

3.1.3 Security Threats and Security Objectives

The security threats identified for the message processing system include the following:

1. Users may gain access to classified messages which they are not cleared to access.

2. Classified messages may (accidentally or deliberately) be delivered to users or agencies who are not cleared to receive them.

3. During processing, information may (accidentally or deliberately) be added whose classification is higher than that of the message, without subsequent adjustment of the message's classification.

4. Users of external agency facilities may try to subvert the system, for example by sending messages containing malevolent code such as computer viruses.

The agency's overall computer security objectives for the system are as follows:

1. To preserve the *confidentiality* of messages – i.e., to ensure that no message is distributed to an individual who is not sufficiently cleared to receive it, nor sent to an agency with a lower classification (no unauthorised disclosure).

2. To preserve the *integrity* of messages – i.e., to ensure that message contents are not accidentally or deliberately changed in transit (no unauthorised modification).

3. To ensure *accountability* of users for their actions – i.e., to ensure that anyone who authorises transmission of a message is identifiable and a record is kept of their actions. This is an important deterrent against deliberate breaches of security.

3.1.4 Conceptual Model of the Security Policy

The following principles underlie the conceptual model of the electronic security policy for the new system:

1. Users are *partitioned* according to their clearance. This applies both to users who are internal to the agency and those in connected external agencies.

2. Confidentiality of information is preserved by controlling the flow of information between user partitions.

3. *Seals* are applied to parts of messages to enable the integrity of their classification and contents to be checked. Any changes to the classification or contents of a message need authorisation before the message can be transferred between partitions.

4. Accountability is enforced by maintaining a *complete audit trail* of system and user actions related to authorisation of messages and attempts to transfer messages between partitions.

The mechanisms which achieve these principles are outlined below.

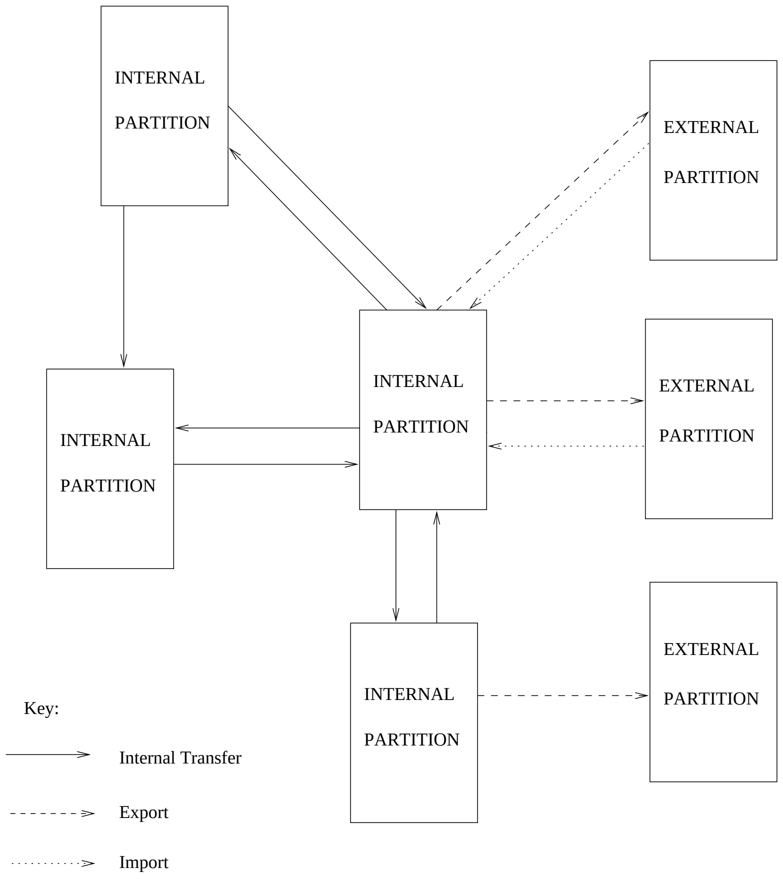

Figure 3.1: Conceptual model of security partitions.

Partitions

Conceptually, the system is divided into a number of *internal partitions* – serving a community of users within the agency who are cleared to access messages passed to (or within) that partition – and a set of *external partitions*, serving other agencies connected to the message processing system (see Fig. 3.1).

A partition can hold messages that are classified up to, and including, the classification of the partition. A user may have access to more than one internal partition, provided of course they have sufficient clearance. Note also that there is not necessarily a physical relationship between the location of users and an internal partition: a single internal partition may be spread over many different physical locations that comprise the distributed system.

Transfers

The operations which move messages from one partition to another are called *transfers*. For precision, transfers between internal partitions will be called *internal transfers*; transfers from internal partitions to external partitions will be called *exports*; and transfers from external partitions to internal partitions will be called *imports*. A non-hierarchical *adjoinment* relation will be used to record how partitions are connected to one another via network gateways. Note that in some cases the flow of information is one-way only (see Fig. 3.1).

Certain constraints will be imposed on the transfer operations by the security policy. In particular, a message will only be transferred from one partition to another if the two partitions adjoin, the receiving partition has sufficient clearance to accept the message, and any changes made to the message have been authorised. Fig. 3.2 illustrates how a message is processed by the system.

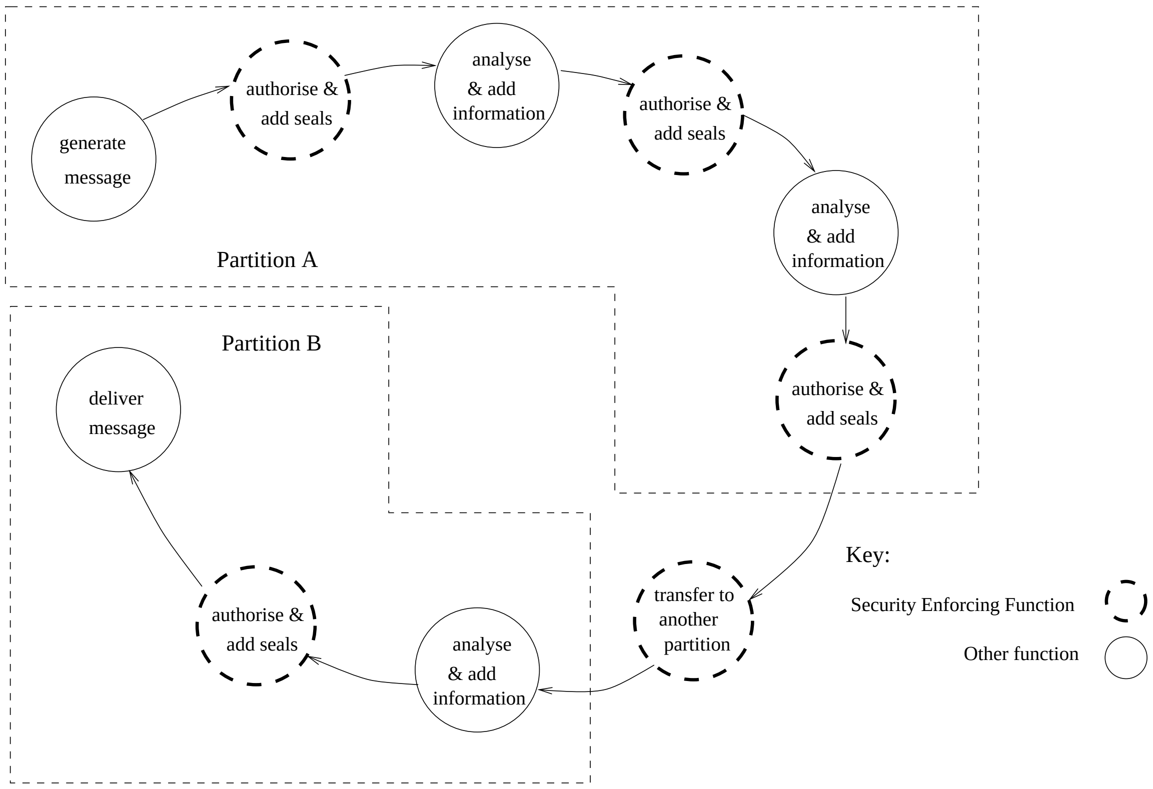

Figure 3.2: Example life-cycle of a message in the message processing system.

Seals

Conceptually, a seal binds the classification of a message part to its contents in a trusted manner. Intuitively, an electronic seal is a kind of encrypted check-sum. The sealing function will be carefully protected from unauthorised use. The essential property of a seal is that authorised users can check the integrity of message parts and their classification by regenerating the seal and checking that it hasn't changed.

Each internal partition has its own protected sealing mechanism. In addition to their use for checking integrity, this SPM uses seals as a mechanisms for maintaining confidentiality, by checking seals before transfer is allowed. This principle is explained in detail in the body of the chapter. In essence, the sealing mechanism provides a *trusted path* between the authoriser and trusted software performing gateway or access control decisions.

Audit trail

Details of each use of a message authorisation operation or transfer operation — successful or unsuccessful — are recorded as part of a security audit trail, including the identity of the authoriser.

3.1.5 The Security Enforcing Functions

The conceptual model of security policy is achieved through four *Security Enforcing Functions*(SEFs), outlined below:

- An *Authorise Message* function, to authorise transfers and apply seals. Authorising requires the user to check the content of the message and to confirm the message is correctly classified. Upon authorisation, seals are added to the message.

- An *Internal Transfer* function, to perform transfers between internal partitions. The function confirms that the message has been sealed and that the destination partition has sufficient clearance to receive the message.

- An *Export* function to perform transfers from internal partitions to external partitions. A confirmation procedure similar to that for Internal Transfer is performed.

- An *Import* function to perform transfers from external partitions to internal partitions. Since in this case the message has been received from an external agency, it is not considered to have been authorised in the required manner. The Import function thus checks that the message contains no viruses, hostile software, etc, before sealing and transferring the message into an internal partition.

The system has many other functions, but the above four are the ones that are concerned with enforcing security.

3.1.6 Specification and Validation of the SPM

In the remainder of this chapter, the Security Policy Model is formally specified in VDM-SL:

- Section 3.2 defines a data model which describes the main system entities at an appropriate level of abstraction.
- Section 3.3 defines the abstract state of the network at any time as consisting of: the conceptual location of messages; the active user sessions; and the complete audit trail of user and system actions. The main security properties are defined as constraints (invariants) on the allowable states.
- In Section 3.4, the four Security Enforcing Functions are modelled as state-changing operations. Exception conditions are used to model abnormal operation, including accidental or deliberate attempts to subvert security.

The SPM is validated in various ways in Section 3.5. In particular, it is shown that the specification is mathematically consistent, the Security Enforcing Functions preserve the desired security properties, and the specification is complete with respect to its input space. Finally Section 3.6 draws some conclusions about the use of specification and proof on this example.

3.2 The Data Model

This section gives mathematical definitions of the main system entities and the relationships between them, including the various static security-enforcing properties of the network. In what follows, *primitive types* are types which will not be defined further here.

3.2.1 Partitions

The primitive type *Partition* will be used to model the set of all possible partitions. The sets of internal and external partitions will be modelled as constants, with declarations:

$intpartns\colon Partition\text{-}\mathbf{set}$
$extpartns\colon Partition\text{-}\mathbf{set}$

The adjoinment relation will be modelled as a binary relation on partitions:

$adjoins\colon Partition \times Partition \to \mathbf{B}$

Thus, $adjoins(p_1, p_2)$ stands for the assertion that messages are physically able to flow from partition p_1 to partition p_2. Note that the configuration of partitions may change from time to time, but that operations for reconfiguring the network are outside the scope of the SPM described here.

3.2.2 Users and User Sessions

The primitive type *UserId* will be used to model the set of identifiers of users. It is assumed that user identifiers are unique and are sufficient to enable a user to be identified unambiguously. (User authentication mechanisms are outside the scope of the SPM described here.) A binary predicate

$hasAccess \colon UserId \times Partition \to \mathbf{B}$

will model the check that a given user has access to a given partition. Note that a user may be able to access multiple partitions (but not simultaneously).

The concept of *sessions* is introduced for periods of use of the system by internal users. Work areas can be shared by a number of users and it is not practical to authenticate users' identities at all times; sessions thus allow an extra level of identification for accountability.

The primitive type *SessionId* will be used to model the set of identifiers of individual user sessions. Conceptually, each session has a unique identifier, together with a record of the partition in which it is being run and the identity of the user who is running it:

$$\begin{array}{lrcl} Session :: & sid & : & SessionId \\ & pid & : & Partition \\ & uid & : & UserId \end{array}$$
$$\textbf{inv}\ s, p, u \triangleq p \in intpartns \wedge hasAccess(u, p)$$

The invariant says that sessions run in internal partitions only, and that some kind of access control is in place to ensure that only users who can access the given partition are able to run sessions there.

3.2.3 Classifications

The primitive type *Classif* will be used to model the set of all possible classifications of messages. In practice, classifications are not simply hierarchical in nature, but have multiple dimensions. The binary predicate

$hasClearance \colon Partition \times Classif \to \mathbf{B}$

will be used to model the relationship between partitions and the classifications of messages they are cleared to receive. The clearance of an individual will be determined implicitly by the partitions they are able to access.

3.2.4 Messages

The central concept of the model is a *message*, which consists of a destination list, a classification, and a set of *message parts*:

$$\begin{array}{lrcl} \mathit{Message} :: & \mathit{destins} & : & \mathit{Destination}\text{-}\mathbf{set} \\ & \mathit{classif} & : & \mathit{Classif} \\ & \mathit{body} & : & \mathit{MessagePart}^* \end{array}$$

A *destination* consists of a user identifier for the intended recipient, together with the partition in which they will receive the message:

$$\begin{array}{lrcl} \mathit{Destination} :: & \mathit{uid} & : & \mathit{UserId} \\ & \mathit{pid} & : & \mathit{Partition} \end{array}$$

The system does not actively check that the intended recipient has access to the destination partition. (Access control is applied at the partition itself.)

The primitive type *Content* is used to model the set of all possible contents of messages. A message part consists of some content, an optional user identifier to note the person who has authorised the content, and an optional seal (explained below):

$$\begin{array}{lrcl} \mathit{MessagePart} :: & \mathit{content} & : & \mathit{Content} \\ & \mathit{authoriser} & : & [\mathit{UserId}] \\ & \mathit{seal} & : & [\mathit{Seal}] \end{array}$$

Creation and processing of the message parts are outside the scope of the SPM. Seals and user identifiers would not be edited under normal circumstances, but the model covers the possibility of malicious editing.

3.2.5 Seals

Seals are applied to individual message parts to enable integrity of their classification, contents and authoriser to be checked. Each of the internal partitions has its own sealing mechanism, which is assumed to be protected in such a way that only authorised users of that partition can access the mechanism, and then only via the Security Enforcing Functions. Intuitively, a seal is a kind of encrypted checksum which binds the classification, content and authoriser to the message part; any attempt to modify these will be detected by regenerating and checking the seal.

The primitive type *Seal* will be used to model the set of all possible seals. The function

$$\mathit{generateSeal}\colon \mathit{Partition} \times \mathit{Classif} \times \mathit{Content} \times \mathit{UserId} \to \mathit{Seal}$$

will be used to model the generation of seals in internal partitions.

The following function checks the integrity of a message part with respect to a given partition and classification:

$$\mathit{hasValidSeal} : \mathit{MessagePart} \times \mathit{Partition} \times \mathit{Classif} \to \mathbf{B}$$

$$\begin{array}{l} \mathit{hasValidSeal}(\mathit{mp}, p, c) \quad \triangleq \quad \mathit{mp.authoriser} \neq \mathbf{nil} \wedge \\ \qquad \mathit{mp.seal} = \mathit{generateSeal}(p, c, \mathit{mp.content}, \mathit{mp.authoriser}) \end{array}$$

Note that the seal may be valid in one partition but not in another. (There will be functions for changing the seals on message parts when messages are transferred from one partition to another.)

The following lemma is a logical consequence of the above definition:

ValidSeal Lemma If a message part's seal is valid, then it is non-nil and an authoriser is identifier for the message part:

$$\begin{array}{l} \forall mp\colon \mathit{MessagePart}, p\colon \mathit{Partition}, c\colon \mathit{Classif} \cdot \\ \quad \mathit{hasValidSeal}(mp, p, c) \;\Rightarrow\; mp.\mathit{seal} \neq \mathbf{nil} \wedge mp.\mathit{authoriser} \neq \mathbf{nil} \end{array}$$

3.2.6 Sealing

The following function models sealing of an individual message part:

$$\mathit{sealMsgPart} : \mathit{Partition} \times \mathit{Classif} \times \mathit{MessagePart} \times \mathit{UserId} \to \mathit{MessagePart}$$

$$\begin{array}{l} \mathit{sealMsgPart}(p, c, mp, u) \;\; \triangleq \;\; \mathbf{if}\ \mathit{hasValidSeal}(mp, p, c)\ \mathbf{then}\ mp \\ \quad \mathbf{else}\ \mathit{mk\text{-}MessagePart}(mp.\mathit{content}, u, \mathit{generateSeal}(p, c, mp.\mathit{content}, u)) \end{array}$$

Note that if message part already has a valid seal then neither it nor the authoriser identifier are changed.

The following function models the sealing of an entire message:

$$\mathit{sealMessage} : \mathit{Partition} \times \mathit{Message} \times \mathit{UserId} \to \mathit{Message}$$

$$\begin{array}{l} \mathit{sealMessage}(p, m, u) \;\; \triangleq \\ \quad \mu(m, \mathit{body} \mapsto [\mathit{sealMsgPart}(p, m.\mathit{classif}, m.\mathit{body}(i), u) \mid i \in \mathbf{inds}\ m.\mathit{body}]) \end{array}$$

The following lemma is a logical consequence of the above definition:

Main Sealing Lemma Sealing a message does not change its destination list, its classification or the content of its parts, and the authoriser field is changed only for message parts without valid seals:

$$\begin{array}{l} \forall p\colon \mathit{Partition}, m\colon \mathit{Message}, u\colon \mathit{UserId} \cdot \\ \quad \mathbf{let}\ m' = \mathit{sealMessage}(p, m, u)\ \mathbf{in} \\ \qquad m'.\mathit{destins} = m.\mathit{destins} \wedge \\ \qquad m'.\mathit{classif} = m.\mathit{classif} \wedge \\ \qquad \mathbf{len}\ m'.\mathit{body} = \mathbf{len}\ m.\mathit{body} \wedge \\ \qquad \forall i \in \mathbf{inds}\ m.\mathit{body} \cdot \mathbf{let}\ mp = m.\mathit{body}(i), mp' = m'.\mathit{body}(i)\ \mathbf{in} \\ \qquad\quad mp'.\mathit{content} = mp.\mathit{content} \wedge \\ \qquad\quad \mathbf{if}\ \mathit{hasValidSeal}(mp, p, m.\mathit{classif}) \\ \qquad\quad \mathbf{then}\ mp'.\mathit{authoriser} = mp.\mathit{authoriser} \\ \qquad\quad \mathbf{else}\ mp'.\mathit{authoriser} = u \end{array}$$

Note that sealed message parts can be inspected without breaking the seal, but any changes to the message part will be detectable. If a message is resealed after message parts have been changed, then the person who authorises the resealing will be identified as the authoriser of the new and changed parts (only).

3.2.7 Changing Seals

The following functions change the seals on a transferred message so that they are valid in its new partition. The functions will be assumed to be protected in such a way that they can be invoked only by the system, and then only at network gateways.

$$\mathit{changeSeal} : \mathit{Partition} \times \mathit{Classif} \times \mathit{MessagePart} \to \mathit{MessagePart}$$

$$\begin{array}{l}\mathit{changeSeal}(p, c, mp) \ \triangleq \\ \quad \mu(mp, \mathit{seal} \mapsto \mathit{generateSeal}(p, c, mp.\mathit{content}, mp.\mathit{authoriser}))\end{array}$$

$$\textbf{pre } mp.\mathit{authoriser} \neq \textbf{nil}$$

$$\mathit{changeSeals} : \mathit{Partition} \times \mathit{Message} \to \mathit{Message}$$

$$\begin{array}{l}\mathit{changeSeals}(p, m) \ \triangleq \\ \quad \mu(m, \mathit{body} \mapsto [\mathit{changeSeal}(p, m.\mathit{classif}, m.\mathit{body}(i)) \mid i \in \textbf{inds } m.\mathit{body}])\end{array}$$

$$\textbf{pre } \forall mp \in \textbf{elems } m.\mathit{body} \cdot mp.\mathit{authoriser} \neq \textbf{nil}$$

The following lemma is a logical consequence of the above definition:

Resealing Lemma Resealing a message does not change its destination list, its classification, nor the content or authoriser fields of its parts:

$$\begin{array}{l}\forall p\colon \mathit{Partition}, m\colon \mathit{Message} \cdot \\ \quad (\forall mp \in \textbf{elems } m.\mathit{body} \cdot mp.\mathit{authoriser} \neq \textbf{nil}) \ \Rightarrow \\ \quad \textbf{let } m' = \mathit{changeSeals}(p, m) \textbf{ in} \\ \qquad m'.\mathit{destins} = m.\mathit{destins} \wedge \\ \qquad m'.\mathit{classif} = m.\mathit{classif} \wedge \\ \qquad \textbf{len } m'.\mathit{body} = \textbf{len } m.\mathit{body} \wedge \\ \qquad \forall i \in \textbf{inds } m.\mathit{body} \cdot \textbf{let } mp = m.\mathit{body}(i), mp' = m'.\mathit{body}(i) \textbf{ in} \\ \qquad\quad mp'.\mathit{content} = mp.\mathit{content} \wedge \\ \qquad\quad mp'.\mathit{authoriser} = mp.\mathit{authoriser}\end{array}$$

3.2.8 Other Integrity Checks

The following predicate checks whether all message parts of a given message have valid seals:

$$\mathit{allSealsAreValid} : \mathit{Message} \times \mathit{Partition} \to \mathbf{B}$$

$$\mathit{allSealsAreValid}(m, p) \ \triangleq \ \forall mp \in \textbf{elems } m.\mathit{body} \cdot \mathit{hasValidSeal}(mp, p, m.\mathit{classif})$$

The following predicate checks that a message has no seals (valid or otherwise):

$hasNoSeals : Message \rightarrow \mathbf{B}$

$hasNoSeals(m) \quad \triangleq \quad \forall mp \in \mathbf{elems}\, m.body \cdot mp.seal = \mathbf{nil}$

The following function strips all seals off the message parts in a given message:

$stripSeals : Message \rightarrow Message$

$stripSeals(m) \quad \triangleq \quad \mu(m, body \mapsto [\mu(m.body(i), seal \mapsto \mathbf{nil}) \mid i \in \mathbf{inds}\, m.body])$

The following lemma is a logical consequence of the above definition:

StripSeals Lemma After applying *stripSeals*, the message has no seals:

$\forall m \colon Message \cdot hasNoSeals(stripSeals(m))$

3.2.9 Content Checks

A primitive predicate

$contentUserChecked \colon Content \times UserId \rightarrow \mathbf{B}$

will model the assertion that the content of a message part has been authorised by the given user. Through the use of seals it will follow that, if this predicate is true, then the message part's content has not subsequently changed in any way.

Similarly, a primitive predicate

$contentAutoChecked \colon Content \rightarrow \mathbf{B}$

will model the assertion that an automated check (e.g. for malicious code) has been applied successfully. This check will be applied to all messages imported from an external partitions. A primitive function

$filterContent \colon Content \rightarrow Content$

will model a function which removes potentially dangerous content (program code, etc) from a message part.

The following functions are used to rebuild a message after its contents have been filtered:

$rebuildMsgPart : MessagePart \rightarrow MessagePart$

$rebuildMsgPart(mp) \quad \triangleq \quad mk\text{-}MessagePart(filterContent(mp.content), \mathbf{nil}, \mathbf{nil})$

$rebuildMessage : Message \rightarrow Message$

$rebuildMessage(m) \quad \triangleq$
$\quad \mu(m, body \mapsto [rebuildMsgPart(mp.body(i)) \mid i \in \mathbf{inds}\, mp.body])$

The following lemma is a logical consequence of the above definitions:

RebuildMessage Lemma Rebuilt messages have no seals:

$\forall m \colon Message \cdot hasNoSeals(rebuildMessage(m))$

3.2.10 Accountability Records

The primitive type *AccRecord* will be used to model the set of all possible *accountability records* which may be stored as part of the system's audit trail.

3.2.11 The Message Pool

The final concept in the data model is that of a *message pool*, which represents the complete collection of messages that are undergoing processing within the network. Message identifiers will be introduced, to simplify modelling of the processing and delivery of messages within the network. Intuitively, a message's attributes may change during processing, but the *identity* of the message will be preserved by each of the Security Enforcing Functions, to allow trace-back.

The primitive type *MsgId* will be used to model the set of all possible message identifiers. The following type will be used to model pools of messages, indexed by the partition in which they reside and their message identifier:

$MessagePool = Partition \xrightarrow{m} (MsgId \xrightarrow{m} Message)$

Note that, a message may be transferred to more than one partition, but during processing there is (conceptually) at most one copy of the message in each partition.

The following function updates message d, with name n, in partition p in the message pool – or adds it, if it didn't already exist:

$$updateMsgPool : MessagePool \times Partition \times MsgId \times Message \to MessagePool$$

$$updateMsgPool(pool, p, n, d) \quad \triangleq \quad pool \dagger \{p \mapsto (pool(p) \dagger \{n \mapsto d\})\}$$

The following lemma is a logical consequence of the definition:

UpdateMessagePool Lemma Apart from the new message *new*, all messages in the updated message pool were already present before the update took place:

$$\begin{array}{l} \mathbf{let}\ pool' = updateMsgPool(pool, to, n, new)\ \mathbf{in} \\ \forall p \in \mathbf{dom}\, pool' \cdot \forall d \in \mathbf{rng}\, pool'(p) \cdot \\ \qquad (p = to \wedge d = new) \vee (p \in \mathbf{dom}\, pool \wedge d \in \mathbf{rng}\, pool(p)) \end{array}$$

3.3 The System State

The system state consists of three state variables: a message pool, representing the conceptual location of messages; a set of currently active sessions; and a sequence of

```
state $SecureNetwork$ of
        $pool : MessagePool$
        $sessions : Session\text{-}\mathbf{set}$
        $audittrail : AccRecord^*$
inv $pool, sessions, audittrail \triangleq$
    $\mathbf{dom}\, pool \subseteq intpartns \cup extpartns \wedge$
    $(\forall p \in \mathbf{dom}\, pool \cap extpartns \cdot \forall m \in \mathbf{rng}\, pool(p) \cdot hasNoSeals(m)) \wedge$
    $\forall p \in \mathbf{dom}\, pool \cap intpartns \cdot \forall m \in \mathbf{rng}\, pool(p) \cdot \forall mp \in \mathbf{elems}\, m.body \cdot$
        $hasValidSeal(mp, p, m.classif) \Rightarrow$
            $hasClearance(p, m.classif) \wedge$
            $contentUserChecked(mp.content, mp.authoriser)$
end
```

Figure 3.3: The state of the secure network, with its important security properties.

accountability records, representing the complete audit trail. The following *security properties* are required to hold at all times:

1. Messages reside only in recognised internal and external partitions.
2. Messages in external partitions have no seals. (Seals should be stripped off messages before they are exported.)
3. If any part of a message in an internal partition has a valid seal, then
   - the partition has clearance to store the message, and
   - the content of that part has been checked by the authorising person and has not subsequently changed.

   This clause formalises the trusted-path property which the sealing mechanism is intended to provide (Section 3.1.4).

The security properties are expressed as an invariant of the state in Figure 3.3.

3.4 Operations Modelling the SEFs

This section gives a formal specification of the four Security Enforcing Functions (SEFs) described in Section 3.1.5 above. Each SEF is modelled as a VDM operation which may change the values of the state variables.

In what follows, and in the subsequent validation, preconditions of operations will have two parts:

1. An *environmental precondition*, which models the important security characteristics of the operation's interface and the conditions under which the operation can be performed. (Certain implementation-level restrictions will not

be modelled here, such as checking that the audit trail recording mechanism is working.)

2. A *precondition for success*, which models the additional conditions which determine whether an attempt has been made to subvert security, for example by attempting to transfer a message to a partition which does not have clearance to receive it.

VDM exception conditions (see p.214 [4]) will be used here to model accidental or deliberate attempts to breach security. An exception is raised whenever the environmental precondition is satisfied but the precondition for success is not. The general form of the specification of the SEFs is thus:

Operation (*inputs*)
ext ...
pre $P \wedge S$
post A_0
errs $FAILURETYPE_1 \colon P \wedge E_1 \rightarrow A_1$
$\vdots$
$FAILURETYPE_n \colon P \wedge E_n \rightarrow A_n$

where P represents the environmental precondition, S represents the precondition for success, A_0 represents the action upon success, and $A_1, \ldots, A_n$ represent the individual actions upon failure.

3.4.1 The Authorise Message Operation

Description: The *AuthoriseMessage* operation is invoked by a user from a session within an internal partition. By authorising a message, the user is taking responsibility for checking the contents of all message parts which did not have a valid seal.

Environmental precondition: The message should reside in the partition in which authorisation takes place, in a currently active session.

Preconditions for success:

1. The partition should have sufficient clearance to store the message. (This check will for example prevent someone who is used to working in multiple partitions from accidentally creating a message in a partition which does not have appropriate clearance.)
2. All addresses in the destination list should refer to the current partition or to an immediately adjoining partition, and the latter should have clearance to receive the message.

$AuthoriseMessage\ (s{:}\ Session, p{:}\ Partition, n{:}\ MsgId)$
ext wr $pool, audittrail$
  rd $sessions$
pre $s \in sessions \wedge s.pid = p \wedge hasAccess(s.uid, p) \wedge$
  $p \in \mathbf{dom}\, pool \wedge n \in \mathbf{dom}\, pool(p) \wedge$
  $sealingAllowed(p, pool(p)(n))$
post let $old = \overleftarrow{pool}(p)(n),\ c = old.classif,\ new = sealMessage(p, old, s.uid)$ **in**
  $(\forall mp \in \mathbf{elems}\, old.body \cdot \neg\, hasValidSeal(mp, p, c) \ \Rightarrow$
    $contentUserChecked(mp.content, mp.authoriser)) \wedge$
  $pool = updateMsgPool(\overleftarrow{pool}, p, n, new) \wedge$
  $audittrail = \overleftarrow{audittrail} \frown [authoriseSuccess(s, p, new)]$
errs $AUTHORISEMESSAGEFAIL$:
    $s \in sessions \wedge s.pid = p \wedge hasAccess(s.uid, p) \wedge$
    $p \in \mathbf{dom}\, pool \wedge n \in \mathbf{dom}\, pool(p) \wedge$
    $\neg\, sealingAllowed(p, pool(p)(n))$
      $\rightarrow\ pool = \overleftarrow{pool} \wedge$
        $audittrail = \overleftarrow{audittrail} \frown [authoriseFailure(s, p, pool(p)(n))]$

Figure 3.4: The operation for authorising a message and adding seals.

Action upon success: Fresh seals are added to all message parts and the audit trail is updated.

Action upon failure: A record of the invalid attempt to authorise a message is added to the audit trail but the message is not changed in any way.

The formal specification of the *AuthoriseMessage* operation is given in Figure 3.4, where

$sealingAllowed : Partition \times Message \rightarrow \mathbf{B}$

$sealingAllowed(p, m) \ \ \triangleq\ \ hasClearance(p, m.classif) \wedge$
  $\forall a \in m.destins \cdot a.pid \neq p$
  $\Rightarrow\ \ adjoins(p, a.pid) \wedge hasClearance(a.pid, m.classif)$

3.4.2 The Internal Transfer Operation

Description: The *InternalTransfer* operation is invoked automatically when a message n arrives at an internal network gateway, from partition *from* to partition *to*.

Environmental precondition: Partitions *from* and *to* should be internal partitions which adjoin, and the message should currently reside in the *from* partition.

Preconditions for success:

1. All seals in the message should be valid with respect to the *from* partition.
2. The *to* partition should appear among the addresses in the message's destination list.
3. To *to* partition should have sufficient clearance to receive the message.

Action upon success: The message is copied across to the new partition, with fresh seals, and the audit trail is updated accordingly.

Action upon failure: A record of the invalid attempt to transfer a message is added to the audit trail.

The formal specification of the *InternalTransfer* operation is given in Figure 3.5, where

$$transferAllowed : Partition \times Partition \times Message \to \mathbf{B}$$

$$transferAllowed(from, to, m) \quad \triangleq \quad allSealsAreValid(m, from) \land$$
$$(\exists a \in m.destins \cdot a.pid = to) \land hasClearance(to, m.classif)$$

3.4.3 The Export Operation

Description: The *Export* operation is invoked automatically when a message *n* arrives at a network gateway from an internal partition *from* to an external partition *to*.

Environmental precondition: Partitions *from* and *to* should be adjoining partitions – *from* internal and *to* external. The message should currently reside in the *from* partition.

Preconditions for success: As for *InternalTransfer*.

Action upon success: The message is copied across to the new partition, with seals removed, and the audit trail is updated.

Action upon failure: A record of the invalid attempt to export a message is added to the audit trail.

The formal specification of the *Export* operation is given in Figure 3.6.

$$\begin{array}{l}
\mathit{InternalTransfer}\ (\mathit{from}\colon \mathit{Partition}, \mathit{to}\colon \mathit{Partition}, n\colon \mathit{MsgId}) \\
\mathbf{ext\ wr}\ \mathit{pool}, \mathit{audittrail} \\
\mathbf{pre}\ \mathit{from} \in \mathbf{dom}\,\mathit{pool} \cap \mathit{intpartns} \wedge \\
\qquad \mathit{to} \in \mathit{intpartns} \wedge \mathit{adjoins}(\mathit{from}, \mathit{to}) \wedge \\
\qquad n \in \mathbf{dom}\,\mathit{pool}(\mathit{from}) \wedge \\
\qquad \mathit{transferAllowed}(\mathit{from}, \mathit{to}, \mathit{pool}(\mathit{from})(n)) \\
\mathbf{post\ let}\ \mathit{new} = \mathit{changeSeals}(\mathit{to}, \overleftarrow{\mathit{pool}}(\mathit{from})(n))\ \mathbf{in} \\
\qquad \mathit{pool} = \mathit{updateMsgPool}(\overleftarrow{\mathit{pool}}, \mathit{to}, n, \mathit{new}) \wedge \\
\qquad \mathit{audittrail} = \overleftarrow{\mathit{audittrail}} \frown [\mathit{transferSuccess}(\mathit{from}, \mathit{to}, n)] \\
\mathbf{errs}\ \mathit{TRANSFERFAIL}\colon \\
\qquad\quad \mathit{from} \in \mathbf{dom}\,\mathit{pool} \cap \mathit{intpartns} \wedge \\
\qquad\quad \mathit{to} \in \mathit{intpartns} \wedge \mathit{adjoins}(\mathit{from}, \mathit{to}) \wedge \\
\qquad\quad n \in \mathbf{dom}\,\mathit{pool}(\mathit{from}) \wedge \\
\qquad\quad \neg\, \mathit{transferAllowed}(\mathit{from}, \mathit{to}, \mathit{pool}(\mathit{from})(n)) \\
\qquad\qquad \rightarrow\ \mathit{pool} = \overleftarrow{\mathit{pool}} \wedge \\
\qquad\qquad\quad\ \ \mathit{audittrail} = \overleftarrow{\mathit{audittrail}} \frown [\mathit{transferFailure}(\mathit{from}, \mathit{to}, n)]
\end{array}$$

Figure 3.5: The operation for copying a message from one internal partition to another.

3.4.4 The Import Operation

Description: The *Import* operation is invoked automatically when a message n arrives at a network gateway from an external partition *from* to an internal partition *to*.

Environmental precondition: Partitions *from* and *to* should be adjoining partitions — *from* external and *to* internal. The message should currently reside in the *from* partition.

Preconditions for success:

1. The *to* partition should appear among the addresses in the message's destination list.
2. To *to* partition should have sufficient clearance to receive the message.
3. The automated check should have been applied successfully to the contents of all message parts.

Action upon success: The message is copied across to the new partition, with its contents filtered to remove any potentially dangerous content and with fresh seals added; the audit trail is updated accordingly.

$Export\ (from\colon Partition, to\colon Partition, n\colon MsgId)$
ext wr $pool, audittrail$
pre $from \in \mathbf{dom}\, pool \cap intpartns\ \wedge$
$\quad to \in extparts \wedge adjoins(from, to)\ \wedge$
$\quad n \in \mathbf{dom}\, pool(from)\ \wedge$
$\quad transferAllowed(from, to, pool(from)(n))$
post let $new = stripSeals(\overleftarrow{pool}(from)(n))$ **in**
$\quad pool = updateMsgPool(\overleftarrow{pool}, to, n, new)\ \wedge$
$\quad audittrail = \overleftarrow{audittrail} \frown [exportSuccess(from, to, n)]$
errs $EXPORTFAIL$:
$\qquad from \in \mathbf{dom}\, pool \cap intpartns \wedge$
$\qquad to \in extparts \wedge adjoins(from, to) \wedge$
$\qquad n \in \mathbf{dom}\, pool(from) \wedge$
$\qquad \neg\, transferAllowed(from, to, pool(from)(n))$
$\qquad\quad \rightarrow\ pool = \overleftarrow{pool} \wedge$
$\qquad\qquad audittrail = \overleftarrow{audittrail} \frown [exportFailure(from, to, n)]$

Figure 3.6: The operation for copying a message from an internal partition to an external partition.

Action upon failure:

1. If the *to* partition does not have sufficient clearance to receive the message, a record of the invalid attempt to import a message is added to the audit trail.
2. If the import check fails, a record of the attempt to import a potentially dangerous message is added to the audit trial.

The formal specification of the *Import* operation is given in Figure 3.7, where

$$importAllowed : Partition \times Partition \times Message \rightarrow \mathbf{B}$$

$$\begin{aligned} &importAllowed(from, to, m) \quad \triangleq \\ &\quad (\exists a \in m.destins \cdot a.pid = to) \wedge hasClearance(to, m.classif) \wedge \\ &\quad \forall mp \in \mathbf{elems}\, m.body \cdot contentAutoChecked(mp.content) \end{aligned}$$

3.5 The Proofs

This section validates the Security Policy Model by showing that the specification is mathematically consistent, the Security Enforcing Functions preserve the security

$Import\ (from\colon Partition, to\colon Partition, n\colon MsgId)$

$\mathbf{ext\ wr}\ pool, audittrail$

$\mathbf{pre}\ from \in \mathbf{dom}\, pool \cap extpartns\ \wedge$
$\quad to \in intparts \wedge adjoins(from, to)\ \wedge$
$\quad n \in \mathbf{dom}\, pool(from)\ \wedge$
$\quad importAllowed(to, pool(from)(n))$

$\mathbf{post\ let}\ new = rebuildMessage(\overleftarrow{pool}(from)(n))\ \mathbf{in}$
$\quad pool = updateMsgPool(\overleftarrow{pool}, to, n, new)\ \wedge$
$\quad audittrail = \overleftarrow{audittrail} \frown [importSuccess(from, to, new)]$

$\mathbf{errs}\ IMPORTTRANSFERFAIL\colon$
$\quad from \in \mathbf{dom}\, pool \cap extpartns \wedge$
$\quad to \in intparts \wedge adjoins(from, to) \wedge$
$\quad n \in \mathbf{dom}\, pool(from) \wedge$
$\quad \neg\, hasClearance(to, pool(from)(n).classif)$
$\quad\quad \rightarrow\ pool = \overleftarrow{pool} \wedge$
$\quad\quad\quad audittrail = \overleftarrow{audittrail} \frown [importTransferFailure(from, to, n)]$

$\quad IMPORTFAIL\colon$
$\quad from \in \mathbf{dom}\, pool \cap extpartns \wedge$
$\quad to \in intparts \wedge adjoins(from, to) \wedge$
$\quad n \in \mathbf{dom}\, pool(from) \wedge$
$\quad hasClearance(to, pool(from)(n).classif) \wedge$
$\quad \neg\, importAllowed(to, pool(from)(n))$
$\quad\quad \rightarrow\ pool = \overleftarrow{pool} \wedge$
$\quad\quad\quad audittrail = \overleftarrow{audittrail} \frown [importFailure(from, to, n)]$

Figure 3.7: The operation for copying a message from an external partition to an internal partition.

properties defined as part of the state invariant, and the specification is complete with respect to its input space.

3.5.1 Consistency Proofs

There are five parts to the proof of mathematical consistency of the model [2]:

1. The specification is syntax and type correct.

2. All function definitions are well formed and agree with the given signatures.

3. All uses of partial functions are well formed, in the sense that the function's arguments are in its domain.

4. All data type invariants are satisfiable (and hence all data types are non-empty).

5. The success or failure of each operation is uniquely determined: i.e., the pre-condition for success and the exception conditions do not overlap. To the best of our knowledge, this condition has not been made explicit in the literature before now. (Strictly, it is a consistency property of the application domain, in which exception conditions have a particular interpretation: it is thus closer to a "proof opportunity" than a proof obligation in the strictest sense of the word.)

The first two parts are straightforward.

For the third part, note that all uses of partial functions in this specification have one of the following forms:

i. $m.body(i)$ where m: $Message$

ii. $pool(p)$ or $pool(p)(n)$ where $pool$: $MessagePool$

iii. $changeSeal(p, c, mp)$

iv. $changeSeals(p, m)$

For (i), it is easy to check that $i \in \mathbf{inds}\, m$ in each case. For (ii), it is easy to check that $p \in \mathbf{dom}\, pool$ and $n \in \mathbf{dom}\, pool(p)$ in each case. For (iii), the only use of *changeSeal* in the specification is in the definition of *changeSeals*, whose precondition guarantees the preconditions of *changeSeal* are satisfied. For (iv), the only use is $changeSeals(to, \overleftarrow{pool}(from)(n))$ in the postcondition of *InternalTransfer*. The precondition of *InternalTransfer* guarantees that, for the message in question, all message parts have valid seals, and hence (by the ValidSeal Lemma in Section 3.2.5) that they all identify an authoriser, as required.

Turning now to data type invariants, the only occurrences in the specification are for *Session* and the state of the system. Both are easily seen to be satisfiable: e.g. the state invariant is satisfied when the message pool is empty.

Given an operation with precondition for success S and given exception conditions $E_1, \ldots, E_n$, proving non-overlap amounts to showing $\neg(S \wedge E_i)$ and $\neg(E_i \wedge E_j)$ for $i \neq j$. For the *AuthoriseMessage*, *InternalTransfer* and *Export* operations, there is a single exception condition and it is of the form $\neg S$, so non-overlap is obvious. For the *Import* operation, there are two exception conditions, of the form $\neg Q$ and $Q \wedge \neg S$ respectively, where $S \Rightarrow Q$. The proofs of the required properties are straightforward.

3.5.2 Preservation of the Security Properties

For each operation, there is a VDM proof obligation to show that, for any state and inputs which satisfy the operation's precondition, there is a corresponding state which satisfies the operation's postcondition. (This is usually called the *satisfiability* proof obligation for operations [2].) For each of the four operations considered here, the values of the post-state are defined explicitly in terms of the pre-state and the inputs, so the proof obligation reduces to showing that the new values preserve the system's state invariant.

Since the message pool is the only state variable mentioned in the state invariant, and since the exception cases of the operations do not actually change the message pool in any way, it suffices to consider only the successful cases of the four operations. Each of the properties defined in Section 3.3 shall be considered in turn below.

Property 1: message location

Property 1 says that all messages reside in internal and external partitions only:

$$\mathbf{dom}\, pool \subseteq intpartns \cup extpartns$$

For each operation, preservation of this property follows easily from the fact that

$$pool = updateMsgPool(\overleftarrow{pool}, p, n, new) \;\Rightarrow\; \mathbf{dom}\, pool \subseteq \mathbf{dom}\, \overleftarrow{pool} \cup \{p\}$$

and the fact that $p \in intpartns \cup extpartns$; the latter is a consequence of the environmental precondition in each case.

Property 2: messages in external partitions

Property 2 says that messages in the external partitions have no seals:

$$\forall p \in \mathbf{dom}\, pool \cap extpartns \cdot \forall m \in \mathbf{rng}\, pool(p) \cdot hasNoSeals(m)$$

To show that this property is preserved, it suffices to consider the *Export* operation only, since the other operations do not affect the messages in the external partitions. Let *to* be the destination partition and let *new* be the exported message. It follows from the UpdateMessagePool lemma that *new* is the only new message in the pool. Since, by the induction hypothesis, all other messages in external partitions have no seals, it suffices to show that *new* has no seals. From the postcondition of *Export* we know that *new* is of the form $stripSeals(m)$ for some message m, so the desired result follows from the StripSeals lemma. The proof is given in detail in Fig. 3.8.

Property 3: messages in internal partitions

Paraphrased, Property 3 says that, for those parts of messages in internal partitions which have a valid seal, then the partition has clearance to store the message and the content has been user-checked:

$$\begin{array}{l}\forall p \in \mathbf{dom}\, pool \cap intpartns \cdot \forall m \in \mathbf{rng}\, pool(p) \cdot \forall mp \in \mathbf{elems}\, m.body \cdot \\ \quad hasValidSeal(mp, p, m.classif) \;\Rightarrow \\ \qquad hasClearance(p, m.classif) \wedge \\ \qquad contentUserChecked(mp.content, mp.authoriser)\end{array}$$

from $\forall p \in \mathbf{dom}\, \overleftarrow{pool} \cap extpartns \cdot \forall m \in \mathbf{rng}\, \overleftarrow{pool}(p) \cdot hasNoSeals(m)$		
1	$new = stripSeals(old)$	post-*Export*
2	$pool = updateMsgPool(\overleftarrow{pool}, to, n, new)$	post-*Export*
3	**from** $p \in \mathbf{dom}\, pool \cap extpartns,\ m \in \mathbf{rng}\, pool(p)$	
3.1	$p \in \mathbf{dom}\, pool$	sets, 3.h1
3.2	$(p = to \wedge m = new) \vee (p \in \mathbf{dom}\, \overleftarrow{pool} \wedge m \in \mathbf{rng}\, \overleftarrow{pool}(p))$	UpdateMessagePool lemma, 2, 3.1, 3.h2
3.3	**from** $p = to,\ m = new$	
3.3.1	$hasNoSeals(stripSeals(old))$	StripSeals lemma
3.3.2	$hasNoSeals(new)$	subs, 1, 3.3.1
	infer $hasNoSeals(m)$	subs, 3.3.h2, 3.3.2
3.4	**from** $p \in \mathbf{dom}\, \overleftarrow{pool},\ m \in \mathbf{rng}\, \overleftarrow{pool}(p)$	
3.4.1	$p \in extpartns$	sets, 3.h1
3.4.2	$p \in \mathbf{dom}\, \overleftarrow{pool} \cap extpartns$	sets, 3.4.h1, 3.4.1
	infer $hasNoSeals(m)$	Induction Hypothesis h1, 3.4.2, 3.4.h2
	infer $hasNoSeals(m)$	cases, 3.2, 3.3, 3.4
infer $\forall p \in \mathbf{dom}\, pool \cap extpartns \cdot \forall m \in \mathbf{rng}\, pool(p) \cdot hasNoSeals(m)$		$\forall$-intro,3

Figure 3.8: Proof of preservation of Property 2 by *Export*.

To show that this property is preserved, it suffices to consider only the operations which affect the messages in the internal partitions: *AuthoriseMessage*, *Internal-Transfer* and *Import*. As for the proof of Property 2 above, it suffices to show that the desired property holds for the *new* message. Specifically, it suffices to show that: the destination partition *to* has sufficient clearance to receive *new*; and that each message part of *new* with a valid seal has had its contents checked by the given authoriser. These properties are proved below for each of the three operations in turn.

AuthoriseMessage:

For the $AuthoriseMessage(s, p, n)$ operation, *new* is the result of adding fresh seals for partition p to the message *old* identified by n. Since sealing does not affect the message's classification, it follows that *old* and *new* have the same classification. Also, the *AuthoriseMessage* operation can be performed only if the *sealingAllowed* test is passed, from which it follows that p must have sufficient clearance to store *old* and hence must have sufficient clearance to store *new*, as desired.

Since, as a result of sealing, all parts of *new* have valid seals, it is necessary to show that all parts have had their content checked by the given autoriser (and have not

```
from $new = sealMessage(p, old, s.uid)$
1     $new.classif = old.classif$                                   Main Sealing Lemma, h1
2     $sealingAllowed(p, old)$                                       pre-$AuthoriseMessage$
3     $hasClearance(p, old.classif)$                                 defn of $sealingAllowed$, 2
4     $hasClearance(p, new.classif)$                                 subs, 1, 3
5     from $i \in \mathbf{inds}\ new.body,\ mp' = new.body(i),$
           $hasValidSeal(mp', p, new.classif)$
          let $mp = old.body(i)$ in
5.1       $mp'.content = mp.content$                                 Main Sealing Lemma, h1, 5.h2
5.2       $\neg hasValidSeal(mp, p, old.classif) \Rightarrow$
              $contentUserChecked(mp.content, s.uid)$
                                                                     post-$AuthoriseMessage$
5.3       from $hasValidSeal(mp, p, old.classif)$
5.3.1         $hasClearance(p, old.classif) \wedge$
              $contentUserChecked(mp.content, mp.authoriser)$
                                                                     Induction Hypothesis, 5.3.h1
5.3.2         $contentUserChecked(mp.content, mp.authoriser)$
                                                                     $\wedge$-elim, 5.3.1
5.3.3         $mp'.authoriser = mp.authoriser$
                                                                     Main Sealing Lemma, h1, 5.h2, 5.3.h1
          infer $contentUserChecked(mp.content, mp'.authoriser)$
                                                                     subs, 5.3.2, 5.3.3
5.4       from $\neg hasValidSeal(mp, p, old.classif)$
5.4.1         $contentUserChecked(mp.content, s.uid)$
                                                                     *modus ponens*, 5.4.h1, 5.2
5.4.2         $mp'.authoriser = s.uid$
                                                                     Main Sealing Lemma, h1, 5.h2, 5.4.h1
          infer $contentUserChecked(mp.content, mp'.authoriser)$
                                                                     subs, 5.4.1, 5.4.2
5.5       $contentUserChecked(mp.content, mp'.authoriser)$          cases, 5.3, 5.4
5.6       $contentUserChecked(mp'.content, mp'.authoriser)$         subs, 5.1, 5.5
5.7       $hasClearance(p, new.classif) \wedge$
              $contentUserChecked(mp'.content, mp'.authoriser)$
                                                                     $\wedge$-intro, 4, 5.6
      infer $hasValidSeal(mp', to, new.classif) \Rightarrow$
              $hasClearance(to, new.classif) \wedge$
              $contentUserChecked(mp'.content, mp'.authoriser)$
                                                                     $\Rightarrow$-intro, 5.7
infer $\forall mp' \in \mathbf{elems}\ new.body \cdot hasValidSeal(mp', p, new.classif) \Rightarrow$
        $hasClearance(p, new.classif) \wedge$
        $contentUserChecked(mp'.content, mp'.authoriser)$            $\forall$-intro', 5
```

Figure 3.9: Proof of preservation of Property 3 by *AuthoriseMessage*.

changed since then). Let mp' be one of the parts of the message *new* and let mp be the corresponding part of message *old*. Since sealing does not affect message contents, it follows that mp and mp' have the same content, hence it suffices to show that mp has been checked. By invoking the *AuthoriseMessage* operation, the operator is taking responsibility for having checked all message parts which did not have a valid seal, so it only remains to consider the parts which have valid seals. But sealing does not change the value of the *content* or *authoriser* fields for such parts, so this case follows directly from the induction hypothesis. This completes the proof for this operation. The proof for the *new* message case is given in detail in Fig. 3.9.

Internal Transfer:

The *InternalTransfer* proof is similar to the *AuthoriseMessage* proof above. Let *new* be the message created by changing the seals on the message *old* from those for the originating partition *from* to those of the destination partition *to*. As before, *old* and *new* must have the same classification. Thus, from $transferAllowed(from, to, old)$, it follows that *to* has sufficient clearance to store *new*.

It also follows from $transferAllowed(from, to, old)$ that all message parts in *old* have valid seals, and hence from Property 3 applied inductively to *old* that all message parts have had their contents checked. Since resealing does not change the contents or authorisers of message parts, it follows that all message parts in *new* have had their contents checked, which completes the proof for this operation. The proof is given in detail in Fig. 3.10.

Import:

The *Import* proof is quite straightforward, since all seals are stripped off the imported message, and so there is essentially nothing to check. The proof is given in detail in Fig. 3.11.

This completes the proof that all operations preserve Property 3.

3.5.3 Completeness Proofs

This section is concerned with the proof that the specification is complete with respect to its input space. Because the modelling is relatively straightforward in this case, there is little to prove. For each SEF, the environmental precondition describes the constraints which are to be imposed on inputs to the SEFs by their interface to the system environment; it thus suffices to show that, for each SEF, the environmental precondition P implies the precondition for success S or one of the exception conditions E_i: i.e.,

$$P \;\Rightarrow\; S \vee E_1 \vee \ldots \vee E_n$$

```
from $new = changeSeals(to, old)$
1    $new.classif = old.classif$                                    Resealing Lemma, h1
2    $transferAllowed(from, to, old)$                               pre-$InternalTransfer$
3    $hasClearance(to, old.classif)$                                defn of $transferAllowed$, 2
4    $hasClearance(to, new.classif)$                                subs, 1, 3
5    from $i \in$ inds $new.body$, $mp' = new.body(i)$,
          $hasValidSeal(mp', to, new.classif)$
         let $mp = old.body(i)$ in
5.1      $mp'.content = mp.content$                                 Resealing Lemma, h1, 5.h2
5.2      $allSealsAreValid(old, from)$                              defn of $transferAllowed$, 2
5.3      $hasValidSeal(mp, from, old.classif)$
                                                                    defn of $allSealsAreValid$, 5.2
5.4      $hasClearance(from, old.classif) \wedge$
             $contentUserChecked(mp.content, mp.authoriser)$
                                                                    Induction Hypothesis, 5.3
5.5      $contentUserChecked(mp.content, mp.authoriser)$            $\wedge$-elim, 5.4
5.6      $mp'.authoriser = mp.authoriser$                           Resealing Lemma, h1, 5.h2
5.7      $contentUserChecked(mp'.content, mp'.authoriser)$
                                                                    subs, 5.1, 5.5, 5.6
5.8      $hasClearance(to, new.classif) \wedge$
             $contentUserChecked(mp'.content, mp'.authoriser)$
                                                                    $\wedge$-intro, 4, 5.7
     infer $hasValidSeal(mp', to, new.classif) \Rightarrow$
             $hasClearance(to, new.classif) \wedge$
             $contentUserChecked(mp'.content, mp'.authoriser)$      $\Rightarrow$-intro, 5.8
infer $\forall mp' \in$ elems $new.body \cdot hasValidSeal(mp', to, new.classif) \Rightarrow$
        $hasClearance(to, new.classif) \wedge$
        $contentUserChecked(mp'.content, mp'.authoriser)$           $\forall$-intro', 5
```

Figure 3.10: Proof of preservation of Property 3 by *InternalTransfer*.

from	$new = rebuildMessage(old)$	
1	$hasNoSeals(new)$	RebuildMessage Lemma, h1
2	$\forall mp \in \mathbf{elems}\, new.body \cdot mp.seal = \mathbf{nil}$	defn of $hasNoSeals$, 1
3	**from** $mp \in \mathbf{elems}\, new.body$	
3.1	$mp.seal = \mathbf{nil}$	$\forall$-elim, 2, 3.h1
3.2	$\neg\, hasValidSeal(mp, to, new.classif)$	ValidSeal Lemma, 3.1
	infer $hasValidSeal(mp', to, new.classif) \Rightarrow hasClearance(to, new.classif) \wedge contentUserChecked(mp'.content, mp'.authoriser)$	$\Rightarrow$-intro', 3.2
infer	$\forall mp' \in \mathbf{elems}\, new.body \cdot hasValidSeal(mp', to, new.classif) \Rightarrow hasClearance(to, new.classif) \wedge contentUserChecked(mp'.content, mp'.authoriser)$	$\forall$-intro', 3

Figure 3.11: Proof of preservation of Property 3 by *Import*.

For each of the operations *AuthoriseMessage*, *InternalTransfer* and *Export*, there is a single exception condition, which is of the form $\neg S$, where S is the precondition for success. In each of these cases, the result thus follows easily from the following propositional tautology

$$P \;\Rightarrow\; S \vee \neg S$$

upon showing that S is well formed, which is straightforward.

For the *Import* operation, there are two exception conditions, of the form $\neg Q$ and $Q \wedge \neg S$ respectively, where S is the precondition for success. The result follows easily from the following propositional tautology:

$$P \;\Rightarrow\; S \vee \neg Q \vee (Q \wedge \neg S)$$

This completes the proof of completeness.

3.6 Conclusions

In conclusion, this chapter has presented a formal Security Policy Model for an electronic message processing and transmission network with multi-level security classification requirements. The VDM-SL specification language was used to define the model. The model was validated by proving that it is mathematically consistent and that it satisfies its required security properties.

A key feature of the Security Policy Model is the use of a sealing mechanism to preserve integrity of messages. The formal specification states precise requirements for how the sealing mechanism should operate. It also explains how the sealing mechanism provides a trusted path between the authoriser and trusted software performing gateway or access control decisions. The proofs confirm in principle that the mechanism achieves its purpose. This in turn conveys a certain degree of assurance that the model is sound.

The main value of the formal model is that it makes the security policy clear and precise. (See Boswell's paper [3] on use of Z for specification and validation of a security policy model for the NATO Air Command and Control System for more discussion.) For the application in question, most of the benefit of the formalisation process was felt to come from being required to make the security policy explicit, and in particular from trying to express exactly what can be inferred from the fact that a seal is valid (Property 3 above).

However, the formalisation did reveal an oversight in an earlier version of the model, which may have had important security implications. The problem was that the earlier model did not explicitly require that messages in external partitions have all seals removed before being imported; as a result, the proof that *Import* preserves Property 3 could not be completed. Upon reflection, it became apparent that the original model was open to "spoofing", whereby an external user with access to a copy of the sealing function could introduce unauthorised messages into the system. Being required to formalise the desired property and perform the proofs thus resulted in the oversight being revealed and the model being improved.

Note in passing that fully formal proofs were constructed and mechanically checked for an earlier version of the specification, using the Mural formal development support environment [4]. To save work, the formal specification was modified so that the four operations shared a common structure, and lemmas were derived which could be proved once and then applied to all four operations. Formal proofs are needed for high degrees of assurance, but rigorous proofs were more appropriate for this chapter, for clarity and ease of understanding. Note however that the amount of additional effort required to construct fully formal proofs is small [2].

Finally, note that the consistency and completeness proof obligations for exception conditions were developed by the first author in the course of this work, and do not appear to have been published before now.

Acknowledgements

The author gratefully acknowledges the substantial contribution to the work reported here by Peter James of Admiral Computing Australia, especially in relation to guidance on requirements for the case study. The idea of using integrity seals as a mechanism for providing a trusted path is due to Les Cook. Thanks are also due to Tom Allen for his earlier input.

3.7 Bibliography

[1] Information Technology Security Evaluation Criteria (ITSEC). Commission of the European Communities, June 1991. Provisional Harmonised Criteria.

[2] J.C. Bicarregui, J.S. Fitzgerald, P.A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide.* FACIT Series. Springer-Verlag, 1994. ISBN no. 3-540-19813-X.

[3] T. Boswell. Specification and validation of a security policy model. In *Proc. 1st Int. Symp. of Formal Methods Europe (FME'93)*, LNCS 670, pages 42–51. Springer Verlag, 1993.

[4] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, second edition, 1990.

[5] C.B. Jones, K.D. Jones, P.A. Lindsay, and R. Moore. *Mural: A Formal Development Support System.* Springer-Verlag, 1991.

[6] C.E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):243–278, Sept 1981.

Index