Segregation with Communication

David Cooper[1] * and Susan Stepney[1]

Logica UK Ltd, Betjeman House, 104 Hills Road, Cambridge, CB2 1LQ, UK
cooperd@logica.com stepneys@logica.com

Abstract. We have developed a general definition of *segregation* in the context of Z system specifications. This definition is general enough to allow multi-way communications between otherwise segregated parties along defined channels. We have an abstract definition of segregation in terms of the traces allowed by systems, a concrete style of specification to ensure segregation (a generalisation of promotion called *multi-promotion*) and a proof that unconstrained multi-promotion is a sufficient condition to ensure segregation.

1 Introduction

We have been working with the National Westminster Development Team (now *platform seven*), proving the correctness of Smartcard applications for electronic commerce. Two of these products have now achieved ITSEC security certification [ITSEC 1996] at the E6 level (the highest level defined). At the time of writing these are the only two products to have achieved this level of certification.

One of the most important security requirements for a smart card operating system is the segregation of applications — ensuring that co-resident applications are kept from interfering with each other (except along clearly defined communication channels). This paper discusses the general mathematical model of segregation we used, and a number of the issues arising from the task of proving that a Z state-and-operations model of a system possesses the required segregation property.

2 Motivating the problem

Although segregation, non-interference, information flow and access control have all been the subject of investigation for a long time, our needs were specific:

- applications must be kept segregated in general, but must be able to communicate with each other along certain defined channels
- communication channels must support communication between multiple (more than two) applications

* current address: Praxis Critical Systems Ltd, 20 Manvers Street, Bath, BA1 1PX
cooperd@praxis-cs.co.uk

- the model of the system under consideration [Stepney & Cooper 2000] would be written in a conventional state-and-operations style in Z
- any segregation property would have to be shown to be possessed by the Z model through *formal proof*

Although much has been published in this area (see, for example, [Bell & Padula 1976], [Rushby 1981], [Goguen & Meseguer 1984], [Bell 1988], [Jacob 1992], [Roscoe 1995], [Gollman 1998], among many others), nothing existing fitted our needs without modification.

3 Trace definition of segregation

We choose to define segregation by considering a system to be a set of allowed traces (sequences of events) [Hoare 1985], [Hoare & He 1998][1].

We bring all of the interesting behaviour of the system out into the visible events, ignoring any internal state that may be used to control the behaviour. This is consistent with the semantics imposed on Z by the refinement rules [Stepney *et al.* 1998], provided the communication events are rich enough to capture inputs, outputs, initialisation and finalisation.

In our model, a system is completely defined by its allowed $TRACE$s. So a system is a (prefix closed) set of $TRACE$s:

$$SYSTEM == \{s : \mathbb{P}\ TRACE \mid (\forall\, t : s; p : TRACE \mid p \ \mathsf{prefix}\ t \bullet p \in s)\}$$

Some systems behave as though they are made up of independent, segregated applications, communicating with each other through defined channels. Our task is to define the set of such systems.

We impose only as much structure on events as necessary. We are defining the concept of segregation between identified *applications*, and so we require a set of application identifiers

$$[A]$$

We want to have control over the communication between applications, and this requires a structure on the data in the events; a collection of named values:

$$[N, V]$$

A communication $EVENT$ consists of the set of applications that are engaging in the event, identifying which of the named values each application can see

$$EVENT == \{n : A \leftrightarrow N; v : N \pfun V \mid n \neq \varnothing \land \mathrm{dom}\, v = \mathrm{ran}\, n\}$$

At least one application is engaging, and the named values are precisely those visible to at least one application.

[1] We do not need to work with a richer model of a system, which could include properties such as failures/divergences, or statistical properties of traces.

We add a dummy event, Φ, representing the view an application has of events it cannot see.

$$\Phi == (\varnothing[A \times N], \varnothing[N \times V])$$

$$EVENT_\phi == EVENT \cup \{ \Phi \}$$

A sequence of $EVENT$s constitutes a $TRACE$.

$$TRACE == \operatorname{seq} EVENT_\phi$$

3.1 A segregated system

An actual system $TRACE$ consists of a sequence of $EVENT$s. We can view this sequence of $EVENT$s from the perspective of each application, seeing only the $EVENT$s engaged by this application, and seeing only the parts of the $EVENT$s visible to this application.

We can choose how much of an $EVENT$ is visible to an application; one choice is:

$$\eta : EVENT_\phi \longrightarrow A \longrightarrow EVENT_\phi$$

$$\forall\, n : A \leftrightarrow N;\ v : N \twoheadrightarrow V;\ a : A \mid (n, v) \in EVENT_\phi \bullet \eta(n, v)\, a = (n \rhd n(\!|\{a\}|\!),\ n(\!|\{a\}|\!) \lhd v)$$

This choice allows applications to see only the named values identified with this application, and to see any applications engaging that have visibility of at least one named value also visible to this application. Other choices are possible (see section 8 for the implications). The remaining discussion of segregation is valid for any choice.

We lift η up to a projection of whole $TRACE$s, dropping invisible $EVENT$s.

$$\pi : TRACE \longrightarrow A \longrightarrow TRACE$$

$$\forall\, t : TRACE;\ a : A \bullet \pi\, t\, a = \{\, i : \operatorname{dom} t \bullet i \mapsto \eta(t\, i)\, a \,\} \upharpoonright EVENT$$

and lift again to sets of $TRACE$s (systems)

$$\Pi : \mathbb{P}\, TRACE \longrightarrow A \leftrightarrow TRACE$$

$$\forall\, s : \mathbb{P}\, TRACE \bullet \Pi\, s = \bigcup \{\, t : s \bullet \pi\, t \,\}$$

We now give the formal definition of being segregated: for a system to be segregated, any collection of application traces derived from the system (that can legally be re-combined) must yield an allowed trace of the system when recombined.

$$SEG == \{\, s : SYSTEM \mid \bigcup((\Pi \fatsemi \Pi^{\sim})(\!|\{s\}|\!)) = s \,\}$$

Intuitively, what does this mean?

If a system behaves as though it consists of independent applications that interact *only* via explicit shared communication events, then if one of its applications exhibits a certain behaviour in one context, it should be able to exhibit the same behaviour in any other consistent context. By "consistent" we mean supplying the same view of shared communication events.

If this is not true, it means that something must be preventing this application from behaving this way, even though all the allowed interfaces are identical. There must be some interference, some back door communication. That is, the system is *not* behaving as independent applications interacting only via explicit shared communication events.

A communications-segregated system has to be quite large: it must at least contain all the individual application behaviours as possible traces; it must also contain the individual behaviours occurring in any order.

4 Using segregation

A definition of segregation in terms of traces is good, but is not immediately applicable to conventional state-and-operations Z models of systems. In a practical, industrial development (such as the development of which this is part [Stepney & Cooper 2000]) one needs specific tools that can be applied to the system specifications being developed. Each small part of the set of tools is relatively straightforward, but when combined needs care and attention to detail. We summarise here the elements needed to turn the abstract segregation definition into a practical tool, and then expand each element in the succeeding sections.

4.1 Multiple models

To make the definition of the segregation property simple we have chosen to work in the world of traces. To make the modelling of the Smartcard system under investigation practical, we have chosen a conventional state-and-operations style of Z specification. These are worlds apart. Although the conceptual step from one to the other is not particularly large or difficult, the practical step needs to be taken, and needs to account for all the messy detail that invariably appears in real systems. Merging these worlds requires a mathematically sound translation.

We developed a number of models, building in progressively more of the computational framework assumed by Z, defining transition functions at each step. This allows us to take a conventional Z specification, extract its equivalent traces model, and impose the requirements of segregation on it.

4.2 Multi-promotion

We developed a Z specification structure that naturally leads to segregated systems. It is a natural extension of promotion to accommodate the simultaneous update of multiple local states; we call it multi-promotion. The expectation is

that if we can constrain the specification of the system *structurally*, it should be easier to prove that a given system is segregated.

4.3 Unwinding theorem

It is useful, in proving a property of systems over sequences of transitions, to prove an equivalent property over individual transitions. This process is generally known as unwinding. We state and prove the unwinding theorem for our definition of segregation. It is, in essence, that

a Z model written as an unconstrained multi-promotion is segregated

This is the justification for all the previous work. Having developed a clear, abstract definition of segregation, we have now proved that, under a mathematically sound and justifiable transformation, a specifically structured specification will possess the segregation property.

5 Multiple models

We have many ways of specifying a system, each suitable for different purposes.

In the various relational formulations, we have a global state Γ, a model state Σ, inputs I, outputs O, and events $EVENT$. (For simplicity we assume that the global inputs and outputs, and the model inputs and outputs, have the same type.)

$$[\Gamma, \Sigma, I, O, EVENT]$$

We use the following models:

event traces: a system is modelled as a set of traces of events: $\mathbb{P}(\text{seq}\, EVENT)$. This is the model used to define segregation.

input–output traces: a system is modelled as a set of traces of input–output pairs: $\mathbb{P}(\text{seq}(I \times O))$

computational model: a system is modelled as a global state transition relation: $\Gamma \times (\text{seq}\, I \times \text{seq}\, O) \leftrightarrow \Gamma \times (\text{seq}\, I \times \text{seq}\, O)$

state transition system: a system is modelled as a state transition relation: $\Sigma \times I \leftrightarrow \Sigma \times O$, a state initialisation $\Gamma \leftrightarrow \Sigma$, an input initialisation $I \leftrightarrow I$, a state finalisation $\Gamma \leftrightarrow \Sigma$, and an output finalisation $O \leftrightarrow O$.

Z model: like the state transition relation model, but using schemas instead of relations: a state *System*, an operation *SystemBhvr*, state initialisation *InitSystem*, identity input initialisation, state finalisation *FinSystem*, and output finalisation $FinOut_+$, or identity. This is the model used to specify the behaviour of our system.

We need ways of moving between these different descriptions. We develop a number of translation functions, as summarised in figure 1 below.

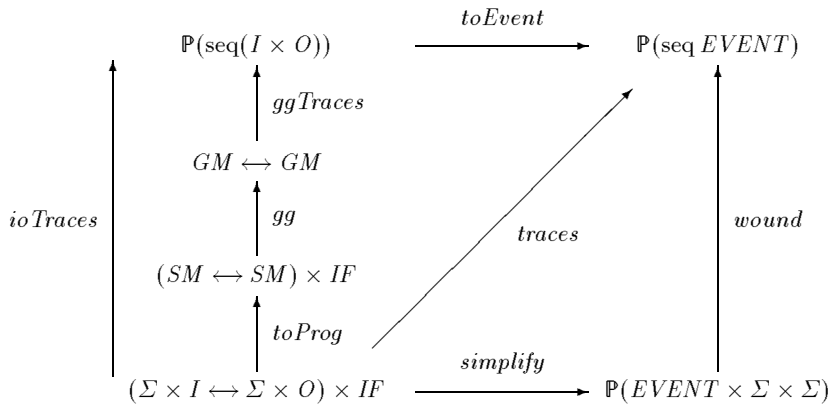

Fig. 1. Summary of the various translation functions

5.1 Traces from the computational model

Z has an implicit computational model used to interpret specifications. This is discussed in some detail in [Woodcock & Davies 1996] and [Stepney *et al.* 1998].

The essential behaviour of a system, and the behaviour we use when deciding whether one system is the refinement of another, is captured in a *relation* between two global states. These global states are deemed to be 'real', in that the elements in them refer to real-world objects and values that can be detected and can be used to test a real system. We define a means of passing back and forth between the global states and the internal, specification states used in Z.

This section develops a function *gg* that maps a Z specification to its essential global-to-global relation. From this, we develop a real-world notion of a system *trace*.

[He *et al.* 1986] is the basis of the theory of Z refinement we use. They make use of general programs written in Dijkstra's guarded command language, but for our purposes we can be more specific. All actions of our system can be represented as a simple sequence of operations, one after the other; no recursion, choice, or non-determinism in operation choice are needed.

5.2 The computational model

As explained in [Woodcock & Davies 1996], we use a system state SM that is rich enough to store the sequence of inputs not yet consumed and the sequence of outputs already produced, as well as the actual system state.

$$SM == \Sigma \times (\text{seq}\, I \times \text{seq}\, O)$$

We are interested in state transitions *sop* that respect the computational model of consuming inputs and producing outputs. Such *sop*s can be written in terms of some input–output state transition *op*:

$$op : \Sigma \times I \leftrightarrow \Sigma \times O \vdash split \fatsemi (op \,||\, id) \fatsemi merge \in SM \leftrightarrow SM$$

(See [Woodcock & Davies 1996], for definitions of *split*, *id*, ||, and *merge*.)

We are interested in the transitive closure of such *sop*s (representing an arbitrary sequence of these operations). Omitting some details, we define the function *toProg*, which takes a relational system description in terms of individual state transitions in the computational model world, and yields the resulting programs in the SM world, by imposing the computational model to yield single state transitions in the SM world, then taking the closure of these to yield the programs.

$$\begin{array}{|l} toProg : (\Sigma \times I \leftrightarrow \Sigma \times O) \longrightarrow (SM \leftrightarrow SM) \\ \hline \forall\, op : \Sigma \times I \leftrightarrow \Sigma \times O \bullet toProg\ op = (split \fatsemi (op \,||\, id) \fatsemi merge)^* \end{array}$$

toProg can also be written explicitly as a constructed set of traces (using extraction functions *in*, *out*, *beforeS*,and *afterS* from the type of *op* in the obvious way):

$$\begin{aligned} toProg\ op = \{\, & \tau : \mathrm{seq}_1\ op \mid \forall\, k : 1 \,..\, \#\tau - 1 \bullet afterS(\tau\, k) = beforeS(tail\, \tau\, k) \bullet \\ & (beforeS(\tau\, 1), (\tau \fatsemi in, \langle\,\rangle)) \mapsto (afterS(\tau(\#\tau)), (\langle\,\rangle, \tau \fatsemi out))\,\} \\ \cup\, & \mathrm{id}\ SM \end{aligned}$$

5.3 Including initialisation and finalisation

We define the set of 'complete' programs by taking the set of programs *allProg op* and adding an initialisation step to the front and a finalisation step to the back. These steps map between the global world and the specification world.

The computational model uses a global structure very similar to the set SM, with a state, input sequence and output sequence. Initialisation maps the global state to the specification state using a state initialisation relation, si; the input sequence to the input sequence using an input initialisation relation, ii; and the output sequence it ignores, giving an empty output sequence in the specification. Finalisation is similar, using sf and of, but ignores the input sequence.

$$GM == \Gamma \times (\mathrm{seq}\ I \times \mathrm{seq}\ O)$$

Define

$$IF == (\Gamma \leftrightarrow \Sigma) \times (I \leftrightarrow I) \times (\Gamma \leftrightarrow \Sigma) \times (O \leftrightarrow O)$$

$$\begin{array}{|l} gg : (SM \leftrightarrow SM) \times IF \longrightarrow GM \leftrightarrow GM \\ \hline \forall\, sopProg : SM \leftrightarrow SM; si, sf : \Gamma \leftrightarrow \Sigma; ii : I \leftrightarrow I; of : O \leftrightarrow O \bullet \\ \quad gg(sopProg, (si, ii, sf, of)) = \\ \qquad (si \,||(\hat{}\, ii \,||(\mathrm{seq}\ O \times \{\langle\rangle\}))) \\ \qquad \fatsemi\, sopProg \fatsemi \\ \qquad (sf^{\sim} \,||((\mathrm{seq}\ I \times \{\langle\rangle\}) \,||\, \hat{}\, of^{\sim})) \end{array}$$

(the $\hat{}$ operator lifts functions on elements to functions on sets of elements.)

gg results in a relation from the initial global state to the final global state.

5.4 Input–Output traces

For segregation we are interested in just the inputs and outputs of a system, ignoring the initial and final states. The function *ggTraces* maps a global-to-global relation to the corresponding set of input/output traces.

$$
\begin{array}{|l}
ggTraces : (GM \leftrightarrow GM) \longrightarrow \mathbb{P}(\mathrm{seq}(I \times O)) \\
\hline
\forall\, r : GM \leftrightarrow GM \bullet \\
\quad ggTraces\, r = \{\, g, g' : \Gamma;\ is, is' : \mathrm{seq}\, I;\ os, os' : \mathrm{seq}\, O \mid \\
\qquad\qquad \#is = \#os' \\
\qquad\qquad \land\ (g, (is, os)) \mapsto (g', (is', os')) \in r \bullet \\
\qquad\quad \{\, i : \mathrm{dom}\, is \bullet i \mapsto (is\, i, os'\, i) \,\} \,\}
\end{array}
$$

Notice that any components in the *gg* relation with differing length input and output sequences have no corresponding input-output trace.

ioTraces maps a state transition relation to a set of input–output traces

$$
\begin{array}{|l}
ioTraces : (\Sigma \times I \leftrightarrow \Sigma \times O) \times IF \longrightarrow \mathbb{P}(\mathrm{seq}(I \times O)) \\
\hline
ioTraces = toProg \fatsemi gg \fatsemi ggTraces
\end{array}
$$

This can be written as explicit sets of traces:

$$
\begin{array}{l}
op : \Sigma \times I \leftrightarrow \Sigma \times O;\ si, sf : \Gamma \leftrightarrow \Sigma;\ ii : I \leftrightarrow I;\ of : O \leftrightarrow O \\
\vdash \\
ioTraces(op, (si, ii, sf, of)) = \\
\quad \{\, \tau : \mathrm{seq}_1\, op;\ g, g' : \Gamma;\ is : \mathrm{seq}\, I;\ os : \mathrm{seq}\, O \mid \\
\qquad\qquad \#is = \#os = \#\tau \\
\qquad\qquad \land\ beforeS(\tau\, 1) \in si(\!|\{g\}|\!) \\
\qquad\qquad \land\ g' \in sf^{\sim}(\!|\{afterS(\tau(\#\tau))\}|\!) \\
\qquad\qquad \land\ (\,\forall\, k : \mathrm{dom}\, \tau \bullet \\
\qquad\qquad\qquad in(\tau k) \in ii(\!|\{is\, k\}|\!) \\
\qquad\qquad\qquad \land\ os\, k \in of^{\sim}(\!|\{out(\tau k)\}|\!) \\
\qquad\qquad\qquad \land\ (k < \#\tau \Rightarrow afterS(\tau\, k) = beforeS(tail\, \tau\, k))\,) \bullet \\
\qquad\quad \{\, i : \mathrm{dom}\, \tau \bullet i \mapsto (is\, i, os\, i) \,\} \,\} \\
\quad \cup \{\langle\rangle\}
\end{array}
$$

5.5 Event traces

The Z computational model is expressed in terms of inputs and outputs, the segregation model in terms of events. We introduce a bijection *asEvent* that relabels inputs and outputs as events, allowing us to move freely between the two alternative representations.

$$
\begin{array}{|l}
asEvent : I \times O \rightarrowtail EVENT
\end{array}
$$

We lift this to the function *toEvent*, which converts from sets of input–output traces to sets of event traces.

$$toEvent : \mathbb{P}(\text{seq}(I \times O)) \longrightarrow \mathbb{P}(\text{seq}\, EVENT)$$

$$toEvent = (\, \lambda\, s : \mathbb{P}(\text{seq}(I \times O)) \bullet (\hat{}\, asEvent)(\!|s|\!)\,)$$

5.6 Mapping from state-and-operations to traces

Finally, we can define the function that takes a system specification written conventionally as state-and-operations, and delivers the equivalent traces model.

$$traces : (\Sigma \times I \leftrightarrow \Sigma \times O) \times IF \longrightarrow \mathbb{P}(\text{seq}\, EVENT)$$

$$traces = ioTraces \fatsemi toEvent$$

This gives us the ability to move formally between our system specification written in a conventional Z style and our definition of segregation written in the traces model.

We have built up this translation function from the theory surrounding Z refinement, which means that we have a good understanding of how this translation is affected by refinement. This is important when relating two system specifications (one a refinement of the other) to the same definition of segregation. This we need to do because in general segregation is not preserved under refinement, and so an abstract system proved to be segregated must be re-proved segregated after refinement. Indeed, we have used this understanding in defining a property of *segregation with respect to* a model (called *segWrt*), which captures the fact that two models are segregated *in the same way*. This is discussed in [Stepney & Cooper 2000].

6 Multi-promotion

6.1 A reminder of single promotion

Promotion is a commonly used technique of structuring a Z specification to aid understanding when the system state consists of a collection of local states, each of which generally changes in isolation (explained in [Barden *et al.* 1994, chapter 19]).

Calling the individual local states *applications*, consider the following simple example:

$$[X, A, C]$$

Each local application has a state (possibly with some invariant predicate) and some locally defined operations taking in an input communication of type C and delivering an output communication of the same type.

ApplState

$$x : X$$

$$\ldots$$

LocalOp

$\Delta ApplState$
$c?, c! : C$

$\ldots$

These are then collected together into a promoted state, where each is labelled by an application name[2]:

PromotedState

$collection : A \longrightarrow ApplState$

Each local application operation can be promoted to an operation on the whole promoted state using a so-called *framing schema*

$\Phi PromotedStateIn$

$\Delta ApplState$
$\Delta PromotedState$
$a? : A$

$collection\ a? = \theta ApplState$
$collection' = collection \oplus \{ a? \mapsto \theta ApplState' \}$

$$PromotedOpIn \mathrel{\widehat=} \exists \Delta ApplState \bullet \Phi PromotedStateIn \land LocalOp$$

We can understand the behaviour of *LocalOp* in the context of a single application state. Having internalised this (and other local operations), all local operations are promoted to the collection in a similar way (via $\Phi PromotedStateIn$) — that one application state is updated according to the local operation, and all other application states don't change.

6.2 Introducing multi-promotion

We extend this structuring to cater for multiple applications changing simultaneously.

First, though, we need to look at some of the features of single promotion. We have identified which application state changes through an input, $a?$. This isn't the only choice — it could be an internal choice of the system based on some system state (such as the "currently selected application"), which we can model by hiding the $a?$:

$$PromotedOpHide \mathrel{\widehat=} \exists \Delta ApplState; a? : A \bullet \Phi PromotedStateIn \land LocalOp$$

[2] Usually, *collection* would be a partial function, and applications could be added and removed by changing its domain. We chose to use a total function and model "absent" applications explicitly, to ease the connection with the segregation definition.

This allows any appropriate application to be the one engaging in the operation — other system constraints may force only one to be appropriate, or the choice may be made non-deterministically. The choice is invisible (unless the resulting state change is visible). Alternatively, the choice can be made an output, $a!$. This behaves like the hidden value (it is the system rather than the user that decides which application state changes), but makes visible which choice was made.

For technical reasons, this was the choice we took in our development.

$$
\begin{array}{|l}
\hline
\Phi PromotedStateOut \\
\Delta ApplState \\
\Delta PromotedState \\
a! : A \\
\hline
collection\ a! = \theta ApplState \\
collection' = collection \oplus \{ a! \mapsto \theta ApplState' \} \\
\hline
\end{array}
$$

$$PromotedOpOut \mathrel{\widehat=} \exists \Delta ApplState \bullet \Phi PromotedStateOut \land LocalOp$$

The inputs and outputs $c?$ and $c!$ pass directly to the (single) application state that changes.

Consider now an extension to allow multiple application states to change:

$$
\begin{array}{|l}
\hline
MPromotedOpOut \\
\Delta PromotedState \\
\alpha! : \mathbb{P}_1\ A \\
c?, c! : C \\
\hline
\alpha! \lhd collection' = \alpha! \lhd collection \\
\forall\ a : \alpha! \bullet \\
\quad \exists \Delta ApplState \bullet \\
\qquad collection\ a = \theta LocalState \\
\qquad \land\ collection'\ a = \theta LocalState' \\
\qquad \land\ LocalOp \\
\hline
\end{array}
$$

A *set* of application names are identified to change. The same options exist in the multiple case as in the single: this can be an input, hidden inside an existential, or an output.

Notice that all local application states are experiencing the same *LocalOp*. This is not a restriction, as it is always possible to harmonise signatures and disjoin all the operations on an application state into a single *LocalOp*, and then use information in the input communication $c?$ or state to select the required operation.

In the case when $\alpha!$ is a singleton set, this formulation reduces to single promotion.

The form just presented is of unconstrained multi-promotion. There are no constraints in the *PromotedState* that affect the ability of individual local applications responding to inputs as they choose. This is why unconstrained multi-promotion fits so naturally with segregation: if all system behaviours are modelled as local operations on local application states, then an unconstrained multi-promotion specifies a system of segregated parts.

Where do the communication channels come in? They appear in the amount of sharing we choose in $\alpha!$, $c?$ and $c!$.

6.3 Defined communication channels

In the Smartcard system we were developing, we had two requirements on communication. First, applications needed to be able to synchronise with others, ensuring that they engaged in an *EVENT* only if specific other applications did (or sometimes, did not) engage. Second, parts of inputs and outputs were sometimes shared.

We met these requirements by expanding on the simple *MPromotedOpOut* in line with the choice of η made in section 3.1.

We modified $\alpha!$, the collection of interacting applications, to be a relation between the interacting applications and the named communications variables they could see.

$$\alpha! : A \leftrightarrow N$$

The inputs and output communications then consists of named values

$$\gamma?, \gamma! : N \nrightarrow V$$

We allowed each application to see a restricted view of $\alpha!$ (it could see the applications that shared at least one named communication variable) and to see those named communication variable identified for this application by $\alpha!$.

This yields

LocalOpWithComms

$\Delta ApplState$
$l\alpha! : A \leftrightarrow N$
$l\gamma?, l\gamma! : N \nrightarrow V$

$\ldots$

MPromotedOpWithComms

$$
\begin{array}{l}
\Delta PromotedState \\
\alpha! : A \leftrightarrow N \\
\gamma?, \gamma! : N \nrightarrow V \\
\hline
\alpha! \neq \varnothing \\
\langle \mathrm{dom}\, \gamma?, \mathrm{dom}\, \gamma! \rangle \ \mathsf{partition}\ \mathrm{ran}\, \alpha! \\
(\mathrm{dom}\, \alpha!) \lhd collection' = (\mathrm{dom}\, \alpha!) \lhd collection \\
\\
\forall\, a : \mathrm{dom}\, \alpha! \bullet \\
\quad \exists\, \Delta ApplState; l\alpha! : A \leftrightarrow N; l\gamma?, l\gamma! : N \nrightarrow V \bullet \\
\qquad l\alpha? = \alpha! \rhd \alpha!(\!|\{a\}|\!) \\
\qquad \land\ l\gamma? = \alpha!(\!|\{a\}|\!) \lhd \gamma? \\
\qquad \land\ l\gamma! = \alpha!(\!|\{a\}|\!) \lhd \gamma! \\
\qquad \land\ collection\ a = \theta LocalState \\
\qquad \land\ collection'\ a = \theta LocalState' \\
\qquad \land\ LocalOp
\end{array}
$$

It is now possible to specify that a local operation will execute only if a specific other application is also executing, by adding the predicate

$$a_1 \in \mathrm{dom}\, l\alpha!$$

or to ensure that at least one other application is executing

$$\#\, \mathrm{dom}\, l\alpha! \geq 2$$

Two applications can exchange a value during execution as follows. Assume one of the named communication variables is *info*. The sending application can set the value of *info*

$$
\begin{array}{l}
info \in \mathrm{dom}\, l\gamma! \\
\land\ l\gamma!\, info = 27
\end{array}
$$

and the receiving application can respond on the basis of the value

$$
\begin{array}{l}
info \in \mathrm{dom}\, l\gamma! \\
\land\ l\gamma!\, info \geq 20 \Rightarrow \ldots
\end{array}
$$

If the receiving application wants to be sure that the value had been set by a specific application, a constraint on $l\alpha!$ can be added.

In these examples there are no constraints in the multi-promotion schema other than those directly related to promotion. In such an *unconstrained multi-promotion* $\alpha!$, $\gamma?$ and $\gamma!$ constitute the communication channels between the applications. There is a clear statement of the information being transmitted — there can be no back-door communication between applications.

If the local constraints are complex, though, it can be hard to determine the actual precondition on a promoted operation. The promoted operation can execute whenever a collection of applications can be found that together have a consistent set of constraints. In practice, this may not be obvious.

7 Unwinding theorem

We have defined a function that extracts from a state-and-operations specification the set of visible $TRACE$s.

$$traces : (\Sigma \times I \leftrightarrow \Sigma \times O) \times IF \longrightarrow SYSTEM$$

This function allows us to formalise the property "our specification $\mathcal{M}$ describes a system that is segregated" as the theorem

$$\vdash traces\, \mathcal{M} \in SEG$$

As it stands, this is quite difficult to prove, because the general definition of SEG is expressed in terms of traces over arbitrary application executions, whereas our specification $\mathcal{M}$ is written as a state-and-operations Z model.

Simplify, based on properties of the model

However, we can make some simplifications, to produce a much simpler sufficient condition. All these simplifications are driven by the particular properties that our model $\mathcal{M}$ has. These properties are:

- initialisation is very simple: no refinement of inputs is needed, so inputs are initialised via the identity; state initialisation is chaotic (it ignores the global state from which initialisation came).
- finalisation is also simple: no refinement of outputs is needed, so outputs are finalised via the identity; all of the state is of interest, so it is finalised via the identity.

We therefore work with $\mathcal{S}$, a simpler representation of $\mathcal{M}$, expressed in terms of $\mathbb{P}(EVENT \times \Sigma \times \Sigma)$, where the dependence on the particular initialisation and finalisation IF has disappeared, and the inputs and outputs have been bundled up into events. The function corresponding to $traces$, that converts the simplified state transition system to a traces description, is called $wound$ (see figure 1).

We prove the **simplification theorem**, that if s is a simplified form of m, then $wound\, s$ gives the same set of traces as $traces\, m$:

$$m : \mathrm{dom}\, traces \vdash traces\, m = wound(simplify\, m)$$

Hence it is sufficient to show that the wound form of our simplified system is segregated.

$$\mathcal{S} == simplify\, \mathcal{M}$$

$$\vdash wound\, \mathcal{S} \in SEG$$

Unwinding

The unwinding step is the heart of our proof; it moves the definition of segregation from the world of traces into the world of simplified state-transition systems like $\mathcal{S}$.

We introduce a set of simplified state transition models, $UNWOUND$, that is a direct analogy of SEG: any application state transition derived from the system by projection must also be a state transition allowed by the system. We prove the **unwinding theorem**, that if a simplified state transition model is in $UNWOUND$, then its traces model has the segregation property:

$$s : UNWOUND \vdash wound\ s \in SEG$$

This proof involves expanding the definition of *wound* to explicitly construct the set of system traces, and then using the properties given in $UNWOUND$, and much tedious algebra, to deduce the properties required by SEG.

Hence it is sufficient to show that the simplified state transition model of our system is in the set $UNWOUND$.

$$\vdash \mathcal{S} \in UNWOUND$$

Labelling

We have moved the segregation proof obligation into the world of general state transitions. We now move into the world a particular kind of state transition: we assume the global state of the system Σ has a structure of labelled local states (where the labels are the application identifiers)

$$\Sigma == A \longrightarrow S$$

We introduce a new set of labelled application systems, $LABELLED$. We prove the **labelling theorem**, that if a simplified state transition model is in $LABELLED$, then it is in $UNWOUND$.

$$s : LABELLED \vdash s \in UNWOUND$$

Hence it is sufficient to show that the simplified state transition model of our system is in the set $LABELLED$.

$$\vdash \mathcal{S} \in LABELLED$$

Promotion

One particular form of a labelled system is a multi-promoted system, a particular way of gluing together labelled local state transitions into a global state transition system. We prove the **promotion theorem**, that such a promoted system is in $LABELLED$.

$$s : PROMOTED \vdash s \in LABELLED$$

Hence it is sufficient to show that the simplified state transition model of our system is in the set $PROMOTED$.

$$\vdash \mathcal{S} \in PROMOTED$$

The set $PROMOTED$ is still expressed in terms of a state transition relation between local labelled states, on events. But it sets the stage for moving from the state transition relation world to the more familiar state-and-operations schemas world.

It is the first time the details of η (the way global events are seen by local applications) appear in the proof. So altering the precise details of the visibility properties of communication channels requires only a small change to the total proof.

Recasting to a schema form

We recast the set $PROMOTED$ into schema form, and show that it is a form of Z unconstrained multi-promotion.

We have reduced the segregation proof obligation to showing that our system is multipromoted. So we now need to show that our system $\mathcal{S}$ can be written as a multi-promoted system. It is time to express our state transition relation in the world of Z schemas.

First, we make a direct translation into Z.

A local application transition has a type like:

$$((A \leftrightarrow N) \times (N \nrightarrow V)) \times S \times S$$

We map this to a schema with a type like:

$$[\, s, s' : S; l\alpha! : A \leftrightarrow N; lc : N \nrightarrow V \,]$$

The $EVENT$-based work we have been doing above makes no distinction between inputs and outputs, so we make an arbitrary division. The local operation schemas do not need to conform to the Z convention for operations, and so the input/output distinction does not need to be made. Here we have mapped the first element of the $EVENT$ pair to $l\alpha!$ and the second to lc.

$$DirectLocalOp \mathrel{\widehat=} [\, s, s' : S; l\alpha! : A \leftrightarrow N; lc : N \nrightarrow V \mid P((l\alpha!, lc), s, s') \,]$$

Here P is some predicate over the state that captures the local operation.

A global application transition has a type like:

$$(A \leftrightarrow N \times N \nrightarrow V) \times (A \rightarrow S) \times (A \rightarrow S)$$

We map this to a schema with a type like:

$$[\, \sigma, \sigma' : A \rightarrow S; g\alpha! : A \leftrightarrow N; gc?, gc! : N \nrightarrow V \,]$$

The global operation, being a normal Z operation, needs to have a before and after state, and inputs and outputs. We have mapped the first element of the *EVENT* pair to $g\alpha!$ and the second to the two variables $gc?$ and $gc!$. We choose these two to not overlap, and between them to cover all of the value of *EVENT*'s second element. We build up the global operation by promoting individual application schemas. The global operation comprises a promotion of local operations:

Global

$$\sigma, \sigma' : A \longrightarrow S$$
$$g\alpha! : A \longleftrightarrow N$$
$$gc?, gc! : N \nrightarrow V$$

$$g\alpha! \neq \varnothing$$
$$\langle \mathrm{dom}\, gc?, \mathrm{dom}\, gc! \rangle \ \mathsf{partition}\ \mathrm{ran}\, g\alpha!$$
$$(\mathrm{dom}\, g\alpha!) \lhd \sigma = (\mathrm{dom}\, g\alpha!) \lhd \sigma'$$

$$\forall a : \mathrm{dom}\, g\alpha! \bullet$$
$$\exists s, s' : S; l\alpha! : A \longleftrightarrow N; lc : N \nrightarrow V \bullet$$
$$s = \sigma\, a$$
$$\land\ s' = \sigma'\, a$$
$$\land\ l\alpha! = g\alpha! \rhd g\alpha!(\!|\{a\}|\!)$$
$$\land\ lc = g\alpha!(\!|\{a\}|\!) \lhd gc? \cup gc!$$
$$\land\ \mathit{DirectLocalOp}$$

Hence it is sufficient to show that our system state and operations model can be written can be written as a *Global* schema.

In fact, we made some further simplifications to *Global*, by instantiating it with the particularly simple value of N in our model. Instead of writing our model in the form of *Global*, we chose a more natural form to express it, and proved that it is equivalent to a *global*-style model [Stepney & Cooper 2000].

Hence we proved our model segregated.

8 Strength of segregation

In the definition of segregation we chose the definition of the projection function η (section 3.1).

At various stages over the course of the development of the model and proof, we found it necessary to change the definition of η. We noticed that changing η had only a small effect on the proof of segregation: it directly affected the structure of a *multi-promoted* state transition system and the corresponding operation schema in Z, but it left the rest of the proof unchanged. It transpires that, in the context of segregation, η determines how much of an event is visible to an operation. Choosing an η that makes more of each event visible means that more systems are classed as segregated, since we have allowed more communication. Had we defined η so that undesirable systems were classed as being segregated, it may not have shown up in the proof.

To address this, we have developed a theory of 'strength of segregation' of η, but do not have the space to go into detail here. In summary, though, we can rank choices of η in a partial order, from the finest (which allows a local application to see no part of any $EVENT$) to the coarsest (which allows a local application to see all of all the $EVENT$s).

The finest segregator is most conservative, in that it permits the minimum number of systems to be classed as segregated. The coarsest segregator permits many systems to be segregated.

We have endeavoured to choose a form of η that is as fine as possible, while still being representative of the actual system being modelled. Thus our choice allows parts of the inputs and outputs to be selectively shared between applications, without forcing all parts to be visible to all applications. If we had chosen a coarser segregator, which revealed all inputs and outputs equally to all applications, we would have been forced to open up wider communication channels between applications than we wanted.

Care must be exercised in choosing η. Too fine, and you will be unable to prove your system segregated. Too coarse, and although you will be able to prove your system is segregated, the form of segregation will be too weak to be useful.

9 Property not preserved by refinement

It is worth noting that segregation as we have defined it is a kind of property not necessarily preserved by refinement. It is possible to specify a system abstractly, prove that such a system is segregated, prove that a more concrete specification is a refinement of the first, but then show that the more concrete specification is not segregated.

The reason for this is that refinement allows non-determinism in the abstract specification to be resolved in any way the implementor chooses in the concrete. One such way may involve using supposedly secret information inside one application to influence the behaviour of another application. This sometimes raises the query of what we mean when we say that the abstract specification was shown to be segregated? If we can exhibit a system that is an implementation of this specification (is a refinement of it) and yet is not itself segregated, how can we say that the abstract specification is segregated?

The definition of segregation is derived from the totality of the system traces allowed by the specification. Being segregated is a property of *all* these traces, and is therefore only necessarily a property of systems that actually exhibit all these traces. Systems that do not exhibit all these traces may be argued to be correct versions of refinements of the specification, but they are not correct versions of the specification itself. We thus take a very constrained view of a specification: it specifies systems that behave in exactly this way; no more, no less.

It is also important to realise the limitations of this. Consider a specification of two applications that output independent values, with no communication

between them. This can be proved to be a segregated system. Consider an actual system that genuinely exhibits all of the specified traces. Such a system would be segregated, by our definition. But this is true even if the system actually chooses its traces from some non-segregated subset of all the traces 99% of the time, and only 1% of the time adds in the full segregated behaviour.

For example, the system could keep the outputs from the two applications in synchrony 99% of the time, and only 1% of the time allow them to non-deterministically diverge. With this information we could reliably (with 99% confidence) predict the output of one application knowing the output of the other. Segregation is a slippery subject, and not to be entered lightly!

10 Conclusions

As part of an industrial project, we have defined a form of *segregation with communication*, in which a set of applications are shown to be kept separated except for defined channels of communications. These channels allow for an arbitrary number of applications to simultaneously engage in sharing information.

We have given this definition in terms of *system traces*, and have also rigorously developed a set of translation functions from conventional Z state-and-operation specifications to system traces.

We have defined a generalisation of promoting a single local state to promoting multiple local states, called *multi-promotion*, and proved an unwinding theorem that

a Z model written as an unconstrained multi-promotion is segregated

Acknowledgements

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

Parts of the work were carried out by Eoin Mc Donnell, Barry Hearn and Andy Newton (all of Logica). We would like to thank Jeremy Jacob and John Clark for their helpful comments and careful review of this work.

References

[Barden *et al.* 1994]
Rosalind Barden, Susan Stepney, and David Cooper. *Z in Practice*. BCS Practitioners Series. Prentice Hall, 1994.

[Bell & Padula 1976]
David E. Bell and Len J. La Padula. Secure computer system: unified exposition and MULTICS. Report ESD-TR-75-306, The MITRE Corporation, March 1976.

[Bell 1988]
D. E. Bell. Concerning "modelling" of computer security. In *Proceedings 1988 IEEE Symposium on Security and Privacy*, pages 8–13. IEEE Computer Society Press, April 1988.

[Goguen & Meseguer 1984]
J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proceedings 1984 IEEE Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society, 1984.

[Gollman 1998]
Dieter Gollman. *Computer Security.* John Wiley, 1998.

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

[Hoare & He 1998]
C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice Hall, 1998.

[Hoare 1985]
C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[ITSEC 1996]
UK IT Security Evaluation and Certification Scheme, issue 3.0. Technical report, UK ITSEC, Cheltenham, December 1996.

[Jacob 1992]
Jeremy L. Jacob. Basic theorems about security. *Journal of Computer Security*, 1(4):385–411, 1992.

[Roscoe 1995]
A. W. Roscoe. CSP and determinism in security modelling. In *Proceedings 1995 IEEE Symposium on Security and Privacy*, pages 114–127. IEEE Computer Society Press, 1995.

[Rushby 1981]
J. M. Rushby. The design and verification of secure systems. In *Proceedings 8th ACM Symposium on Operating System Principles*, December 1981.

[Stepney & Cooper 2000]
Susan Stepney and David Cooper. Formal methods for industrial products. (These proceedings), 2000.

[Stepney *et al.* 1998]
Susan Stepney, David Cooper, and Jim Woodcock. More powerful Z data refinement: pushing the state of the art in industrial refinement. In Jonathan P. Bowen, Andreas Fett, and Michael G. Hinchey, editors, *ZUM'98: 11th International Conference of Z Users, Berlin 1998*, volume 1493 of *Lecture Notes in Computer Science*, pages 284–307. Springer Verlag, 1998.

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