Security Modelling in Z and HOL

Roger Bishop Jones

ICL Defence Systems

bool ::= T | F

Given a state consisting of one highly classified and one lowly classified object:

STATE[p]
high, low :bool

can we specify loosely an operation on the state which does not result in any information transfer from ‘high’ to ‘low’?

ΛSTATE[p]
STATE, STATE’

?

It is easy enough to give a specific operation satisfying this requirement, but to capture the requirement loosely we have to use a loose specification outside of the schema, e.g.:

f: STATE → STATE

∀s¹ s²:STATE• (s¹.low = s².low)
⇒ ((f s¹).low = (f s²).low)

We could then write our schema:

ΛSTATE[p]
STATE, STATE’

θSTATE’ = f θSTATE

but since all the work has been done in the specification of ‘f’ the use of the schema appears superfluous.

Note that in the axiomatic definition of f, the requirement is expressed as a property of f, but this property has not itself been given a name.

It is therefore not possible to express in the object language the claim that some other explicitly defined function has this property.

For example the following function has the required property:

$$g: STATE \rightarrow STATE$$

$$\forall s : STATE \bullet g\ s = s$$

but we cannot state this in Z without restating the original property (though it can be said in the metalanguage).

To enable such correctness propositions to be expressed we must give a name to the property itself as follows:

$$secure : \mathbb{P}\ (STATE \rightarrow STATE)$$

$$f \in secure \Leftrightarrow$$
$$\forall s^1\ s^2 : STATE \bullet (s^1.low = s^2.low)$$
$$\Rightarrow (f\ s^1).low = (f\ s^2).low$$

The conjecture that ‘g’ satisfies this specification can now be expressed:

$$\vdash ?\ g \in secure$$

If we define a further requirement:

$$\begin{array}{|l} \quad \text{safe} : \mathbb{P}\,(\text{STATE} \rightarrow \text{STATE}) \\ \hline \quad f \in \text{safe} \Leftrightarrow \\ \quad \forall s^1\ s^2 : \text{STATE} \bullet\ (s^1.\text{high} = s^2.\text{high}) \\ \qquad\qquad \Rightarrow (f\ s^1).\text{high} = (f\ s^2).\text{high} \\ \end{array}$$

Then the combination of these two requirements:

$$\begin{array}{|l} \quad \text{no\_flow} : \mathbb{P}\,(\text{STATE} \rightarrow \text{STATE}) \\ \hline \quad \text{no\_flow} = \text{secure} \cap \text{safe} \\ \end{array}$$

may be regarded as a REFINEMENT of the original specification "secure".

That it is a refinement can be expressed in the object language as the conjecture:

$$\vdash ?\ \text{no\_flow} \subseteq \text{secure}$$

Note that here refinement is defined as a relationship between specifications which is distinct from the relationship between a specification and an implementation.

SPECIFYING OPERATIONS AS FUNCTIONS

Type of *Object*
AUTO
Type of *Specification*
$\mathbb{P}$ **AUTO**
Type of *Operation*
$\mathbf{IN} \times \mathbf{STATE} \rightarrow \mathbf{STATE} \times \mathbf{OUT}$
$\subseteq \mathbb{P}(\text{IN} \times \text{STATE} \times \text{STATE} \times \text{OUT})$
Type of *Specification of* Operation
$\mathbb{P}\ (\mathbf{IN} \times \mathbf{STATE} \rightarrow \mathbf{STATE} \times \mathbf{OUT})$
Type of *Non-Deterministic* Operation
$\mathbf{IN} \times \mathbf{STATE} \rightarrow \mathbb{P}^1\ (\mathbf{STATE} \times \mathbf{OUT})$
Type of *Specification of Non-Deterministic* Operation
$\mathbb{P}\ (\mathbf{IN} \times \mathbf{STATE} \rightarrow \mathbb{P}^1\ (\mathbf{STATE} \times \mathbf{OUT}))$
Type of *Partial* Operation
$\mathbf{IN} \times \mathbf{STATE} \nrightarrow \mathbf{STATE} \times \mathbf{OUT}$
Type of *Specification of Partial* Operation
$\mathbb{P}\ (\mathbf{IN} \times \mathbf{STATE} \nrightarrow \mathbf{STATE} \times \mathbf{OUT})$
Type of *Partial Non-Deterministic* Operation
$\mathbf{IN} \times \mathbf{STATE} \nrightarrow \mathbb{P}^1\ (\mathbf{STATE} \times \mathbf{OUT})$
Type of *Specification of Partial Non-Deterministic* Operation
$\mathbb{P}\ (\mathbf{IN} \times \mathbf{STATE} \nrightarrow \mathbb{P}^1\ (\mathbf{STATE} \times \mathbf{OUT}))$

Z SCHEMAS INTERPRETED AS OPERATIONS

Until the publication of Spivey's book "understanding Z" no account was available of how schemas are to be interpreted as specifications of operations.

Spivey gives an account of a satisfaction relationship between schemas and implementations which can be formalised within Z as follows.

Let us consider this with reference to schemas describing the secure operations discussed above.

The type of a schema describing an operation over STATE is:

```
SOPTYPE == ℙ ΛSTATE
```

According to Spivey this is a loose specification of a non-deterministic partial operation (in the general case). It may therefore be interpreted as an entity of type:

```
FTYPE ==  STATE ⇸ ℙ¹ STATE
```

```
INTTYPE ==      ℙ (STATE ⇸ ℙ¹ STATE)
```

A formal account of this interpretation would therefore be a map from OPTYPE to INTTYPE:

```
MAPTYPE == OPTYPE → INTTYPE
```

```
_satisfies_ : FTYPE ↔ SOPTYPE
_____________
∀SOPTYPE:S, FTYPE:f•
      f satisfies S ⇔
            ∀s:STATE•
                  f s ⊆ {S | θ STATE = s • θ STATE'}
```

note that:

```
f satisfies S ∧ g satisfies S
      ⇒ (f merge g) satisfies S

where (f merge g) x = f x ∪ g x
```

```
ML
new_theory ‘form167‘;;
new_parent‘pf‘;;
loadf‘infra‘;;
```

‘STATE‘‘S‘[p]

high:bool, low :bool

T

---

secure : (STATE → STATE) → bool

---

f ∈ secure ⇔
∀s¹ s²:STATE• (s¹.S_low = s².S_low)
⇒ ((f s¹).S_low = (f s²).S_low)

---

g: STATE → STATE

∀s :STATE• g s = s

ML

```
set_goal([],"g ∈ secure");;
e (REWRITE_TAC[secure_THM;X_2e_DEF;g_THM]);;
let g_secure_THM = save_top_thm ‘g_secure_THM‘;;
```

---

safe : (STATE → STATE) → bool

---

f ∈ safe ⇔
∀s¹ s²:STATE• (s¹.S_high = s².S_high)
⇒ ((f s¹).S_high = (f s²).S_high)

ML

```
set_goal([],"g ∈ safe");;
e (REWRITE_TAC[safe_THM;X_2e_DEF;g_THM]);;
let g_safe_THM = save_top_thm ‘g_safe_THM‘;;
```

---

no_flow : (STATE → STATE) → bool

---

no_flow = secure ∩ safe

```
ML
set_goal([],"∀f•f ∈ no_flow ⇔ f ∈ secure ∧ f ∈ safe");;
e (REWRITE_TAC[no_flow_THM; ∩_DEF; ∈_DEF]);;
let no_flow_THM1 = save_top_thm ‘no_flow_THM1‘;;
```

```
ML
set_goal([],"g ∈ no_flow");;
e (REWRITE_TAC[no_flow_THM1; g_secure_THM; g_safe_THM]);;
let g_no_flow_THM = save_top_thm ‘g_no_flow_THM‘;;
```

```
ML
set_goal([],"no_flow ⊆ secure");;
e (REWRITE_TAC[no_flow_THM; ∩_DEF; ⊂_DEF; ∈_DEF]
      THEN TAUT_TAC);;
let no_flow_secure_THM = save_top_thm ‘no_flow_secure_THM‘;;
```

1. SECURE SYSTEM DESIGN USING SECURITY KERNEL

The secure system is just a function over the state which has the property ‘secure’. We show that such a system can be constructed from two subsystems, one of which is a trusted security kernel, and the other of which is an untrusted user program. The trustedness of the kernel is represented by a specificaiton for the kernel which it must satisfy if the total system is to be secure. The untrustedness of the user program is reflected in its trivial specification.

The design consists of a set of subcomponent specifications and a construction showing how the system is built from the subcomponents.

1.1. The user program specification

The user program is any function which operates on the state.

```
      USER_PROGRAM_TYPE == STATE → STATE
```

```
ML
let USER_PROGRAM_TYPE = ":STATE → STATE";;
```

```
      user_program_specification : ℙ USER_PROGRAM_TYPE
──────────────────
      user_program_specification = {f:USER_PROGRAM_TYPE}
```

user_program_specification : ˆUSER_PROGRAM_TYPE → bool

---

user_program_specification = λf:ˆUSER_PROGRAM_TYPE•T

1.2. The Kernel Specification

The kernel mediates between the user program and the secure data store.

KERNEL_TYPE == USER_PROGRAM_TYPE × bool → (STATE → STATE)

kernel_specification : ℙ KERNEL_TYPE

---

k ∈ kernel_specification ⇔
∀f:USER_PROGRAM_TYPE, b:bool• k(f,b) ∈ secure

ML
```
let KERNEL_TYPE = ":ˆUSER_PROGRAM_TYPE × bool → (STATE → STATE)";;
```

kernel_specification :
ˆKERNEL_TYPE → bool

---

k ∈ kernel_specification ⇔
∀(f:ˆUSER_PROGRAM_TYPE) (b:bool)• k(f,b) ∈ secure

i.e. a kernel is something which can run an untrusted program at any classification without permitting a breach of security.

1.3. The Construction

The system as a whole consists of a security kernel running some application at some clearance. Let us leave the classification loosely undefined.

user_clearance:bool

user_clearance:bool

T

The construction is a function which takes a security kernel and a user program and yields a secure system:

```
secure_system_construction : (KERNEL_TYPE × (STATE → STATE))
                           → (STATE → STATE)
```

```
∀k:KERNEL_TYPE, u:STATE → STATE•
        secure_system_construction (k,u) = k(u,user_clearance)
```

```
secure_system_construction : (^KERNEL_TYPE × ^USER_PROGRAM_TYPE)
                           → (STATE → STATE)
```

```
∀(k:^KERNEL_TYPE) (u:^USER_PROGRAM_TYPE)•
        secure_system_construction (k,u) = k(u,user_clearance)
```

The correctness of this design is the conjecture that this construction will always yield secure system when applied to subcomponents which meet their specifications.

```
?⊢ ∀k:KERNEL_TYPE, u:USER_PROGRAM_TYPE•
            u ∈ user_program_specification
      ∧     k ∈ kernel_specification
      ⇒     secure_system_construction(k,u) ∈ secure
```

```
ML
set_goal([],"∀(k:^KERNEL_TYPE) (u:^USER_PROGRAM_TYPE)•
             u ∈ user_program_specification
      ∧      k ∈ kernel_specification
      ⇒      secure_system_construction(k,u) ∈ secure");;
e (REWRITE_TAC[kernel_specification_THM; secure_system_construction_THM]
      THEN REPEAT STRIP_TAC THEN ASM_REWRITE_TAC[]);;
let design_correct_THM = save_top_thm ‘design_correct_THM‘;;
```

1.4. An Implementation of the Kernel

The design correctness was a very trivial result, because of the way the kernel specification was written. This may look like cheating, but we now show an example of a security kernel and prove that it meets the specification.

The kernel filters the current state before passing it on to the user program, and then filters the state computed by the application before using it to update the real state. These two filtering operations represent the placement of constraints on read and on write which are dependent on the clearance of the user program.

Filtering isnt strict;y possible because of the primitive structure of the state.

We define a function which takes two STATEs and a classification and merges the two states taking values dominated by the classification from the first state and other values from the second.

```
_dominates_ : bool ↔ bool
─────────────
∀x:bool• x dominates x ∧ T dominates F
```

```
dominates: bool → bool → bool
─────────────
∀(x:bool)(y:bool)• dominates x y ⇔ y ⇒ x
```

```
ML
print_theory‘form167‘;;
```

The Theory form167
Parents -- HOL pf
Types -- ":STATE"
Constants --
IS_STATE ":bool × bool → bool"
REP_STATE ":STATE → bool × bool"
ABS_STATE ":bool × bool → STATE" S_high ":STATE → bool"
S_low ":STATE → bool" secure ":(STATE → STATE) → bool"
g ":STATE → STATE" safe ":(STATE → STATE) → bool"
no_flow ":(STATE → STATE) → bool"
user_program_specification ":(STATE → STATE) → bool"
kernel_specification
":((STATE → STATE) × bool → (STATE → STATE)) → bool"
user_clearance ":bool"
secure_system_construction
":((STATE → STATE) × bool → (STATE → STATE)) × (STATE → STATE) →
(STATE → STATE)"
Curried Infixes -- dominates ":bool → (bool → bool)"
Axioms --
STATE_AXIOM
⊢ ∃rep•
(∀x' x''• (rep x' = rep x'') ⇒ (x' = x'')) ∧
(∀x• p_or_choice IS_STATE x = (∃x'• x = rep x'))

Definitions --
REP_STATE
⊢ REP_STATE =
(μrep•
(∀x' x''• (rep x' = rep x'') ⇒ (x' = x'')) ∧
(∀x• p_or_choice IS_STATE x = (∃x'• x = rep x')))
ABS_STATE ⊢ ∀x• ABS_STATE x = (μx'• x = REP_STATE x')

Theorems --
IS_STATE_THM ⊢ ∀high low• IS_STATE(high,low)
STATE_assorted_THMS
⊢ (∀a a'• (REP_STATE a = REP_STATE a') = (a = a')) ∧
(∀r• p_or_choice IS_STATE r = (∃a• r = REP_STATE a)) ∧
(∀r r'•
p_or_choice IS_STATE r ⇒
p_or_choice IS_STATE r' ⇒
((ABS_STATE r = ABS_STATE r') = (r = r'))) ∧
(∀a• ∃r• (a = ABS_STATE r) ∧ p_or_choice IS_STATE r) ∧
(∀a• ABS_STATE(REP_STATE a) = a) ∧
(∀r• p_or_choice IS_STATE r = (REP_STATE(ABS_STATE r) = r))
STATE_IS_REP_THM ⊢ ∀x• IS_STATE(S_high x,S_low x)
ABS_REP_STATE_THM ⊢ ∀x• ABS_STATE(S_high x,S_low x) = x
S_high_THM ⊢ ∀high low• S_high(ABS_STATE(high,low)) = high

S_low_THM ⊢ ∀high low• S_low(ABS_STATE(high,low)) = low

secure_THM
⊢ ∀f•
f ∈ secure ⇔
(∀s¹ s²•
(s¹ . S_low = s² . S_low) ⇒
((f s¹) . S_low = (f s²) . S_low))

g_THM ⊢ ∀s• g s = s

g_secure_THM ⊢ g ∈ secure

safe_THM
⊢ ∀f•
f ∈ safe ⇔
(∀s¹ s²•
(s¹ . S_high = s² . S_high) ⇒
((f s¹) . S_high = (f s²) . S_high))

g_safe_THM ⊢ g ∈ safe

no_flow_THM ⊢ no_flow = secure ∩ safe

no_flow_THM1 ⊢ ∀f• f ∈ no_flow ⇔ f ∈ secure ∧ f ∈ safe

g_no_flow_THM ⊢ g ∈ no_flow

no_flow_secure_THM ⊢ no_flow ⊆ secure

user_program_specification_THM
⊢ user_program_specification = (λf• T)

kernel_specification_THM
⊢ ∀k• k ∈ kernel_specification ⇔ (∀f b• (k(f,b)) ∈ secure)

user_clearance_THM ⊢ T

secure_system_construction_THM
⊢ ∀k u• secure_system_construction(k,u) = k(u,user_clearance)

design_correct_THM
⊢ ∀k u•
u ∈ user_program_specification ∧ k ∈ kernel_specification ⇒
(secure_system_construction(k,u)) ∈ secure

dominates_THM ⊢ ∀x y• x dominates y ⇔ y ⇒ x