On the Refinement of Non-interference

John Graham-Cumming*  
J. W. Sanders  
Oxford University Computing Laboratory  
Programming Research Group  
8-11 Keble Road  
Oxford, OX1 3QD

Abstract

It is known that functional refinement does not preserve the security properties of a system. We propose a solution by giving a trace-based method for specifying the security properties of a system and a method which ensures that this security is preserved under refinement.

We include an example to illustrate the use of our definitions and make use of non-interference (as defined in our notation).

1 Introduction

In the “formal methods” approach to system design, a mathematical language (rather than natural language or a programming language) is used to describe the system at each stage of its development. The system’s functional requirements are captured in a (formal) specification and an implementation is developed hierarchically by steps which preserve functional behaviour.

The advantages and disadvantages of that approach are well documented (see [5] and [11]). Because of the large investment necessary—an investment in time and expertise—it seems, at present, that formal methods are most suited to the design of safety-critical, high integrity and secure systems (see [9]) where the investment is justified. Yet it is in just such systems where the use of (functional) development appears to fail: it is possible to specify a design that exhibits a given form of security and to transform that specification, using standard correctness-preserving laws, into one that violates that form of security; whilst the functional behaviour has been preserved, the security has not. This fact is well known, see for example [4] and [6].

We must, therefore, be sceptical about designing a secure system by starting with a specification embodying a “secure design” (see, for example, [1] and [8]) and using standard refinements. In this paper we take one type of security as a case study and make a positive proposal for development of systems that display it, by restricting the type of refinement allowed. The type of security we consider is non-interference between users of a system who interact through either shared store or communicating processes (or agents). Our aim is to strengthen the current refinement laws so that if users are non-interfering in a specification then they are still non-interfering in any refinement of it.

We define a security specification formalism. We hope that this formalism can be used to define a wide range of security policies, and that it will form part of a methodology for specifying and developing secure computer systems.

2 A Security Specification Formalism

The most common security property to have been formalised is Multi-Level Security (MLS). MLS has its origins in paper-based security in military establishments, where security is defined by a number of ordered levels. As an example, consider the descending sequence of levels: Top Secret, Secret and Unclassified. Both users and objects (files etc.) are associated with one of these levels; users have clearances and objects classifications. It is a basic assumption of MLS that users at a particular level of security clearance are allowed access solely to information below that level. This policy can be enforced by asking that the constraints no read-up and

*EMAIL: jgc@prg.oxford.ac.uk
no write-down are observed. No read-up says that users (people in the paper model; called subjects in general) cannot read information which is classified at a level higher than the user's clearance. Similarly, no write-down prevents high level users passing information to users with lower security clearance than themselves.

An attempt to generalise MLS was made by Goguen & Meseguer, [2, 3], by the introduction of the notion of non-interference. They hoped to produce a general method of specifying security policies, including MLS. Rushby, [12], produced a similar notion of non-interference, which he described by saying:

We say that there is no flow of information from one user to another, or that the first user is non-interfering with the second, if the results seen by the second are completely unaffected by the presence or absence of operations issued by the first.

It is this informal description that we take as a starting point. Whereas Rushby and Goguen & Meseguer used deterministic state machines as a foundation, we use action systems (or abstract data-types). As Sutherland, see [15], pointed out non-interference is a sufficient condition for the absence of information flow.

Central to our definition, and other definitions of security, is the equivalence relation \( \approx \), which is used to determine which traces (histories of operations), or sometimes states, of a system are indistinguishable by a user \( u \) (when thinking of MLS we usually write \( \approx_l \) for equivalent at level \( l \)). In Rushby's and Goguen & Meseguer's work this relation is simply equality of outputs of the system that the user \( v \) could generate; we wish to abstract from a distinction between input and output. In other work this relation has variously been instantiated for particular systems or left completely undefined.

We define \( \approx_u \), informally, by saying

**Definition 1:** Two traces, \( s \) and \( t \), of a system are \( v \)-equivalent (\( s \approx_v t \), or indistinguishable to \( v \)), for user \( v \), if no sequence of operations \( v \) can perform, starting from either trace, enables him to distinguish them. \( \square \)

At this stage we have said little about the definition of a system; that is we have not restricted ourselves to deterministic systems. Additionally our model of security makes no mention of probability or time; both these considerations are left for future examination, and as we have not considered them we do not make any assumptions about security properties which embody them.

We define a security policy as a relation on the traces of a system. These relations, \( P_v \), specify which traces are to be indistinguishable to user \( v \).

**Definition 2:** Given a set of relations, \( P_v \), for each user \( v \), on traces. We say that a system is secure iff \( \forall v \in P_v \subseteq \approx_v \).

\( \square \)

### 2.1 Non-interference

Our notion of non-interference is standard. The major difference between it and those of [3, 12] is that we test equivalence using the relation \( \approx_v \), defined above.

**Definition 3:** A user \( u \) is non-interfering with user \( v \) if \( v \) cannot observe any change in the state of the system if \( u \)'s operations are removed from any trace. Using a function \( \text{purge}_u \), to remove \( u \)'s operations while leaving the other elements of the trace unchanged, we define: \( u \) is non-interfering with \( v \) (\( u \not\approx v \)) iff \( \forall t : \tau S \cdot t \approx_v \text{purge}_u t \).

The set \( \tau S \) consists of sequences of operations (sometimes called histories or execution sequences) of the system recording the history of events (operations) that have occurred; its definition depends on our model of a system and will be made plain below. It is important to note that in general the set \( \tau S \) will not consist of every sequence of operations.

Thinking of purge as a relation on traces the condition for non-interference (\( u \not\approx v \)) is \( |_u \subseteq \approx_v \).

**Proposition 1:** In general, \( \not\approx \) is neither transitive nor symmetric. \( \square \)

The relation \( \not\approx \) can be thought of as the security policy; we consequently say a system is secure if the system is non-interfering with respect to the relation.

### 2.2 Systems

The definition of security is independent of the system it is being implemented on. We define below the type of system we use for the purposes of this paper. It is important to realise that the choice of formalism used to describe the functional properties of the system under consideration does not affect the description of security properties. We have chosen to use Z (see [14]), but
could equally have chosen another (for example, see [7] and [10]).

2.2.1 Systems

A system, $S$, is defined to have:

- A set of States $S$
- A set of Initial States $S_0$, where $S_0 \subseteq S$
- A set of Users $U$, (a user is a set of operations)

We write $S = (S, S_0, U)$.

Each user is identified by the set of operations he may perform, his interface to the system. We write $u.\text{op}$ for a typical operation that $u$ may perform, i.e. a member of $u : U$. An operation $u.\text{op}$ that starts with the system in some state $s_0$, accepts input $\alpha?$ provided by user $u$, delivers output $\beta!$ and leaves the system in state $s$ is denoted

$$u.\text{op} (s_0, \alpha?, \beta!, s).$$

When convenient, drop either or both of the input/output arguments.

Each $u.\text{op}$ is performed by user $u$ alone. Operations are defined as predicates on states and hence may be nondeterministic and partial. An important concept is the precondition of operation $u.\text{op}$ denoted $\text{pre } u.\text{op}$.

$$\text{pre } u.\text{op} (s_0, \alpha?, \beta!, s)$$

is an internal trace of $S$, leaving the system in state $s$.

We start by defining, $\approx$, an equivalence relation on states of the system such that $r \approx s$ if no user can distinguish between states $r$ and $s$. Two system states are equivalent: $r \approx s$ iff for each user-indexed operation $u.\text{op}$,

$$\text{pre } u.\text{op} (r_0, \alpha?) \equiv \text{pre } u.\text{op} (s_0, \alpha?)$$
$$u.\text{op} (r_0, \alpha?, \beta!; r) \equiv u.\text{op} (s_0, \alpha?, \beta!; s)$$

for some states $r, s$. A system whose distinct states are inequivalent is called fully abstract.

An internal trace records the evolution of a system through states, user-indexed operations and consequent input and output. Its users are not, however, privy to the system state and observe only the names of their own operations and the input and output they produce. Because we are interested in behaviour of the system from user's viewpoints we define a system trace (or, in full, an external system trace) to be obtained from an internal system trace by concealing system states. The set of system traces of $S$ is denoted $\tau S$ and defined:

$$\tau S = \{h(t) \mid t \text{ is an internal trace of } S\}.$$  

The state concealing function, $h$, which distributes through concatenation, $\tau$, is defined by giving its value on each singleton trace:

$$h(w.\text{op}(s_0, \alpha?, \beta!, s)) \equiv (w.\text{op}(\alpha?, \beta!)).$$

The domain of $h$ is the set of internal traces. A user $v$ engages in $v.\text{op}'s$ and can only observe the restriction of a system trace to the set of $v$-indexed operations:

$$\{v.\text{op}_1(\alpha_1?, \beta_1!), \ldots, v.\text{op}_n(\alpha_n?, \beta_n!)\}.$$  

2.2.3 Non-interference Defined

To complete the definition of non-interference given by 3 we define the function $\text{purge}_w$, which removes from a system trace all operations performed by user $w$ and show how $\approx_w$ is defined.

Purge distributes through concatenation and is defined for a singleton as:

$$\langle w.\text{op}(\alpha?, \beta!) \rangle_w \equiv \emptyset,$$

$$\text{if } w = u,$$

$$\equiv \langle w.\text{op}(\alpha?, \beta!) \rangle,$$

$$\text{if } w \neq u.$$  

Two system traces are equivalent from the viewpoint of user $w$ if they allow the same subsequent sequence of
u-indexed operations, input and output. Thus, for any 
\( u : U \) and any system traces \( s \) and \( t \), we define \( s \) to be \( v \)-equivalent to \( t \), \( s \approx_v t \), iff for any sequence \( r \) of operations that \( v \) can perform,

\[ (s \circ r \in rS) \equiv (t \circ r \in rS). \]

Note that it does not suffice to let \( r \) be a singleton, or of any bounded length, in that definition. As a consequence the definition is difficult to verify and to reason about.

A user \( v \) views the state of the system as \( v \)-equivalence classes of system traces; in that way he automatically obtains a fully abstract model of the system from his viewpoint. Both \( \approx \) and \( \approx_v \) can be thought of either as relations on system states, or on traces of the system.

**Proposition 2:**
- \( \approx \) and \( \approx_v \) are equivalence relations, on both traces and states.
- \( \approx \subseteq \approx_v \), for any user \( v \).
- \( \forall s, t : rS \cdot s \approx_s t \Rightarrow s \circ r \approx_v t \circ r \), where \( s' \) is a trace consisting solely of operations that \( v \) can perform.

Using the notation \([t]\) to represent the set of states that can result from performing trace \( t \) starting with the system in one of its initial states, we have the following proposition which relates the instantiations of \( \approx \) and \( \approx_v \) on states and on traces.

**Proposition 3:**
- for any traces \( s, t : rS \), \( s \approx_t t \) iff \( \forall a : [s] \cdot \forall b : [t] \cdot a \approx b \)
- for any traces \( s, t : rS \), \( s \approx_v t \) iff \( \forall a : [s] \cdot \forall b : [t] \cdot a \approx_v b \)

2.3 Example
Consider a system which has a pair of users, \( v \) and \( g \), and a collection of operations for set manipulation. We show a refinement of the system in which user \( g \), the administrator's task is garbage collection. The garbage collector is necessary because we implement the abstract set inefficiently as a sequence. Clearly the garbage collector must be non-interfering with the user of the set calculator. This example illustrates that a non-interfering system cannot be partitioned into distinct systems, one for each user; and the necessity of having an equivalence relation over states.

The abstract system has four operations on finite sets of some type \( E \) : insert an element; remove an element; clear the set; test inclusion of an element in the set. The single user \( v \) can perform all of them.

We are using the notation \( Z \) to describe the operations. Each schema, below, has two parts: the first describing the types of any variables used and the second giving a predicate on these variables. When describing the change in value of a variable caused by an operation we use a prime (') as a suffix to denote the final state. The suffixes ? and ! denote input and output respectively.

Our set calculator has a global variable \( set \), a finite set of elements from \( E \). The initialisation operation makes \( set \) empty.

\[
\begin{align*}
\text{Init} & \\
set &: F \ E \\
set' &= \emptyset
\end{align*}
\]

As does \( v.\text{Clear} \): \( v.\text{Clear} \equiv \text{Init} \). Two operations add and remove elements from the set.

\[
\begin{align*}
\text{v.Insert} & \\
set, set' &: F \ E \\
? & : E \\
set' &= set \cup \{?\}
\end{align*}
\]

\[
\begin{align*}
\text{v.Remove} & \\
set, set' &: F \ E \\
? & : E \\
set' &= set \setminus \{?\}
\end{align*}
\]

\( v \) can examine the set using the operation \( v.\text{Element} \) to test for the presence of a particular element.

\[
\begin{align*}
\text{v.Element} & \\
set &: F \ E \\
? & : E \\
g & : B \\
g! &= (? \in set)
\end{align*}
\]
In this abstract description the garbage collector has little to do:

\[
g, \text{Compact} \quad \frac{\text{set, set' : } \mathcal{F} \ E}{\text{set' = set}}
\]

In the abstract system user \( g \) does not change the state and so does not interfere with \( v \). In the concrete system, however, the garbage collection operation is needed. User \( g \) changes the sequence which represents \( v \)'s state and we require that \( v \) cannot detect the change caused by compaction. We have the security policy \( g \not\rightarrow v \).

\( \{(), (v, \text{Insert}(e))\) and \( (v, \text{Insert}(e), g, \text{Compact}) \) are all system traces, for some \( e \in E \). From the viewpoint of user \( g \) the system is in equivalent states after having performed any of them; whilst the last two leave the system in equivalent states from the point of view of user \( v \):

\[
(\{() \approx_g (v, \text{Insert}(e)) \approx_g (v, \text{Insert}(e), g, \text{Compact})).
\]

After \( v \) inserts an element, however, its state is no longer \( v \)-equivalent because the operation \( v, \text{Element} \) will reveal that there is an element, namely \( e \), in the set:

\[
(\{() \not\approx_v (v, \text{Insert}(e)).
\]

### 3 Refinement

In this section we show how to perform refinement from an abstract secure system to a concrete system preserving not only functionality but security. Of course by security we mean that users are non-interfering (as defined in section 2.2.3).

Suppose that \( A = (A, A\text{Init}, U) \) is an abstract system and \( C = (C, C\text{Init}, U) \) is a concrete system. Typical rules that ensure \( C \) refines \( A \) (downwards simulation, [13]) presume the existence of a refinement relation \( \phi \) satisfying,

\[
\begin{align*}
\text{CInit} & \Rightarrow A\text{Init}; \phi & (1) \\
\text{pre u.aop}; \phi & \Rightarrow \text{pre u.cop} & (2) \\
\text{pre u.aop} < (\phi; \text{u.cop}) & \Rightarrow \text{u.aop}; \phi. & (3)
\end{align*}
\]

Where \( \text{u.aop} \) and \( \text{u.cop} \) are corresponding abstract and concrete operations.

The following theorem demonstrates a simple rule for secure refinement. We use the notation \( \approx^s_v \) when we wish to emphasise that we are considering the abstract instantiation of the equivalence relation; similarly we use the superscript \( e \) to denote the concrete system. \([t]_u \) is the equivalence class of trace \( t \) generated by the relation \( \approx_v \).

**Theorem 1:** If \( \phi \) preserves \( \approx_v \), and the abstract system is secure, then the concrete system is secure.

**Proof:** \( t \approx^s_v t|_u \), since the abstract system is secure, so \( t \approx^s_v t|_u \) in the concrete system, since \( \phi \) preserves \( \approx_v \). Hence the concrete system is secure.

This is, however, too weak and difficult to work with. We imagine a map from abstract \( \approx_v \)-classes to concrete \( \approx_u \)-classes, which is defined:

\[
\pi^s_\phi; \theta_v \equiv \phi; \pi^s_v.
\]

We use \( \theta_v \) to give a sufficient condition for the preservation of security.

**Lemma 1:** If \( \theta_v \) is a function then:

\[
\theta_v ([s]^s_v) = [s]^s_u
\]

for any trace \( s \). Proof is a simple induction using the three laws of refinement, 1, 2, and 3.

**Theorem 2:** If for any \( v \) such that there exists \( v \) such that \( u \not\rightarrow v, \theta_v \), as defined by 4, is a function, and the abstract system is secure, then the concrete system is secure.

**Proof:** Since the abstract system is secure we have, \( u \not\rightarrow v \Rightarrow t \approx^s_v t|_u \). In terms of \( \approx_v \)-classes, that becomes:

\[
u \not\rightarrow v \Rightarrow [t]^s_v = [t|_u]^s_u.
\]

Since \( \theta_v \) is a function we have:

\[
u \not\rightarrow v \Rightarrow \theta_v ([t]^s_v) = \theta_v ([t|_u]^s_u).
\]

and so, since \( \theta_v \) relates corresponding equivalence classes:

\[
u \not\rightarrow v \Rightarrow [t]^s_v = [t|_u]^s_u
\]

Hence, \( u \not\rightarrow v \Rightarrow t \approx^s_v t|_u \) in the concrete system. So the concrete system is secure.

The condition of the theorem is not, however, necessary.
4 Example

Returning to the garbage collector, we make a functional refinement (data refinement) by changing the representation of set from a set to a sequence of elements drawn from $E$.

The initialisation operation sets the new global variable $seq$ to be the empty sequence.

\[
\text{CInit}
\begin{align*}
\text{seq} & : \text{seq } E \\
\text{seq} & = ()
\end{align*}
\]

The clear operation changes accordingly: $v.\text{CClear} \triangleq \text{CInit}$. To insert we simply concatenate the element to be inserted onto the end of the sequence.

\[
\text{v.CInsert}
\begin{align*}
\text{seq, seq}' & : \text{seq } E \\
x & : E \\
\text{seq}' & = \text{seq} \text{ } \uparrow \text{ (range seq \{} x? \text{\})}
\end{align*}
\]

Deleting an element requires the removal of all occurrences of that element from the sequence.

\[
\text{v.CRemove}
\begin{align*}
\text{seq, seq}' & : \text{seq } E \\
x & : E \\
\text{seq}' & = \text{seq} \text{ } \uparrow \text{ (range seq \{} x? \text{\})}
\end{align*}
\]

To check for the presence of an element we count the number of occurrences of it in the sequence. The operator $\downarrow$ counts the number of times an element occurs in a sequence.

\[
\text{v.CElement?}
\begin{align*}
\text{seq} & : \text{seq } E \\
x & : E \\
y ! & : \text{B} \\
y ! & = (\text{seq} \downarrow x? > 0)
\end{align*}
\]

The compaction operation now changes the state by removing multiple entries of an element of $E$ from the sequence, saving "space."

\[
\text{g.Compact}
\begin{align*}
\text{seq, seq}' & : \text{seq } E \\
\text{range seq} & = \text{range seq'} \\
\forall a : \text{seq } \& \text{seq}' & | a = 1
\end{align*}
\]

The abstract and concrete systems are related by the refinement relation:

\[
\begin{align*}
\phi & : \text{set } \subseteq \text{E} \\
\text{seq} & : \text{seq } E \\
\text{set} & = \text{range seq}
\end{align*}
\]

We wish to prove that the sequence based concrete system is a functional refinement of the set based abstract system and that it is secure. We regard it as evident that the abstract system is secure: see 5.1.

- **Initial Conditions** First consider the initial conditions $\text{Init}$ and $\text{CInit}$ and prove condition 1. We know $s \text{ CInit } ()$, since the only initial concrete state is $()$. Similarly $s \text{ Init } \emptyset$. But $\emptyset \phi ()$, from the definition of the refinement relation $\phi$. So $\text{Init}; \phi = \text{CInit}$ and hence condition 1 holds.

- **Pre-conditions** Condition 2 automatically holds since all the operations we are considering are total.

- **Operations** To prove functional refinement we need only prove the simplified form of 3: $\forall \text{ cop } w. w. \text{cop} \Rightarrow w. \text{cop}; \phi$ for each of the abstract operations and its corresponding concrete manifestation.

Each operation satisfies condition 3 and as an example of the proof we demonstrate that $v.\text{CInsert}$ refines $v.\text{Insert}$.

U is inserting $t \in E$. From the definition of $\phi$, $v.\text{Insert}$, and $v.\text{CInsert}$:

\[
\begin{align*}
\phi & ; v.\text{CInsert}(t) \{ c \uparrow (t) \mid \text{range } c = s \} & \text{and} & \phi & ; v.\text{Insert}(t) \{ c \mid \text{range } c = (s \cup \{t\}) \}. \\
\phi & ; v.\text{CInsert}(t) = v.\text{Insert}(t) ; \phi.
\end{align*}
\]

And so the concrete system is a functional refinement of the abstract set calculator.

All that remains is to prove security, as defined by the policy $g \not\rightarrow v$. We do this by examining $\theta_v$.

Using the lemma 1 we know that equivalent states of the concrete system are mapped, by $\pi_v^+$, to the same $\equiv_v$-class. The refinement relation, $\phi$, relates a set to any sequence whose range is that set. The equivalent sequences are those with the same range, since none of the $v.\text{op}'s$ distinguish between single and multiple entries of an element. Hence $\phi; \pi_v^+$ is a function.

Now consider $\pi_v^{-1}$. We require that it is a function. Hence we require:
\[
(s \approx_v^u t) \land (a \approx_s^u b) \Rightarrow a \approx_b^u b,
\]
which is true if
\[
(s \approx_v^u t) \land (a \approx_s^u b) \Rightarrow a \approx_b^u b.
\]

This system has this property, since \( s \approx_v^u t \) means that \( v \) cannot distinguish between the two traces and \( a \approx_s^u b \) means that no user can distinguish between the two states. There is only one non-\( v \cdot \text{op} \), \( g' \text{Compact} \), which is total and has no inputs or outputs.

So \( \theta_v \) is a function and by the theorem above the concrete system is secure.

## 5 Winding and Unwinding

The results of section 3 require that the abstract system be secure. In the example of section 4 we quite reasonably assumed that to be true. However in general to prove security requires checking:

\[
V + s \approx_v^u \Rightarrow t\approx_v^u (t\approx_v^u)\]

for all \( t \in r \cdot S \); a lengthy task. We ease that burden with an Unwinding Theorem.

The unwinding conditions are, for a system \( S \),

\[
\begin{align*}
    u \not\approx_v^u v & \quad \Rightarrow \quad t \approx_v^u t\approx_v^u (u\cdot\text{op}) \quad (5) \\
    s \approx_v^u t & \quad \Rightarrow \quad s\approx_v^u (w\cdot\text{op}) \approx_v^u t\approx_v^u (w\cdot\text{op}) \quad (6)
\end{align*}
\]

**Theorem 1:** If, in system \( S \), conditions 5 and 6 hold then it is secure.

**Proof:** The proof is by induction on the length of a trace \( t \).

\( \text{(case } t = () \text{)} \)

This case is trivial from the definition of \( \approx_v^u \) since \( () \approx_v^u () \).

\( \text{(case } t \neq () \text{)} \)

The induction hypothesis is: \( u \not\approx_v^u v \Rightarrow t \approx_v^u t\mid_u \).

Consider a command \( w\cdot\text{op} \) as two cases:

\( \text{(case } w = u \text{)} \)

\[
\begin{align*}
    u \not\approx_v^u v & \quad \Rightarrow \quad t \approx_v^u (t\approx_v^u (w\cdot\text{op}) \land t \approx_v^u t\mid_u) \\
    & \quad \Rightarrow \quad t\approx_v^u (w\cdot\text{op}) \approx_v^u t\mid_u \\
    & \quad \Rightarrow \quad t\approx_v^u (t\approx_v^u (w\cdot\text{op}))
\end{align*}
\]

\( \text{(case } w \neq u \text{)} \)

\[
\begin{align*}
    u \not\approx_v^u v & \quad \Rightarrow \quad t \approx_v^u t\mid_u \\
    & \quad \Rightarrow \quad t\approx_v^u (w\cdot\text{op}) \approx_v^u t\mid_u (w\cdot\text{op}) \\
    & \quad \Rightarrow \quad t\approx_v^u (w\cdot\text{op}) \approx_v^u (t\approx_v^u (w\cdot\text{op}))\mid_u
\end{align*}
\]

Hence \( S \) is secure. \( \Box \)

Condition 5 is not only sufficient, to prove security, but necessary, as shown by the following lemma.

**Lemma 1:** In a secure system, 5 holds.

**Proof:** Consider a command \( u\cdot\text{op} \):

\[
\begin{align*}
    u \not\approx_v^u v & \quad \Rightarrow \quad t\approx_v^u (u\cdot\text{op}) \approx_v^u (t\approx_v^u (u\cdot\text{op}))\mid_u \\
    & \quad \Rightarrow \quad t\approx_v^u (u\cdot\text{op}) \approx_v^u t\mid_u
\end{align*}
\]

Condition 6, however, is not necessary. Consider a system with two users \( u \) and \( v \). The system state is a pair of numbers. Each of the users has access to one of them. A user can set his number, shift its binary representation right one bit or read its parity. The system is non-interfering because the users are entirely separated. Suppose we want to prove that this system is a secure refinement of some other system with the policy \( u \not\approx_v^u v \) and \( v \not\approx_v^u u \).

In the concrete system, user \( u \) sees the numbers 1 and 3 as the same, because they have the same parity. But after a shift right they become 0 and 1, which have opposite parities. In this case condition 6 fails even though the system is obviously non-interfering.

### 5.1 Unwinding the Garbage Collector

To prove that the abstract garbage collector is secure, we use the unwinding theorem. The conditions we have to prove are:

\[
\begin{align*}
    s \approx_v^u (g' \text{Compact}) \\
    s \approx_v^u t & \quad \Rightarrow \quad s\approx_v^u (v\cdot\text{op}) \approx_v^u t\approx_v^u (v\cdot\text{op})
\end{align*}
\]

The second condition is derived from unwinding condition 6 by examining the proof; we only need prove 6 for users \( v \) for which there exist \( u \) such that \( u \not\approx_v^u v \) and only need consider operations \( w\cdot\text{op} \) with \( w \not= v \). Hence, the second condition above holds by proposition 2, for \( \approx_v^u \).

It is clear that:

\[
(a \approx_v s) \land (b \approx_v^u (s\approx_v^u (g' \text{Compact}))) \Rightarrow a \approx_b^u b
\]

because two sets are equivalent if they cannot be distinguished by the \( v\cdot\text{op} 's. \) The \( v\cdot\text{op} 's distinguish between distinct sets. The post-condition of \( g' \text{Compact} \) is:

\[
\text{set} = \text{set}'.
\]

So, by proposition 3 the first unwinding condition holds. Hence the abstract system is secure.
6 Conclusion and Future Work

We have shown the importance of two things: specifying security separately from the functional properties of a system and the need for extra proof rules if conventional forms of refinement are to be used in developing correct secure systems.

We have demonstrated a method for specifying security properties and given a proof rule to help in the construction of a secure system (the Unwinding Theorem). We have shown a refinement law for the preservation of non-interference and demonstrated its use.

This work has also been considered in the light of CSP and action systems with success. CSP has composition operators (|| and ⊥, for example) and it is interesting to see how security properties compose under them. This area is being investigated further and it is hoped to produce algebraic rules for use in the construction of secure systems.

The Unwinding Theorem given here is restricted to non-interference; a more general version can be proved for a class of policies. It is hoped to report on these developments at a later date.

References