High Assurance Software

Non-interference through Determinism

A. W. Roscoe J. C. P. Woodcock L. Wulf

Oxford University Computing Laboratory
Parks Road, Wolfson Building, Oxford OX1 3QD, UK

Abstract. The standard approach to the specification of a secure system is to present a (usually state-based) abstract security model separately from the specification of the system's functional requirements, and establishing a correspondence between the two specifications. This complex treatment has resulted in development methods distinct from those usually advocated for general applications.
We provide a novel and intellectually satisfying formulation of security properties in a process algebraic framework, and show that these are preserved under refinement. We relate the results to a more familiar state-based (Z) specification methodology. There are efficient algorithms for verifying our security properties using model checking.

Keywords
Security, Non-interference, Formal methods, Process algebra,
Determinism, Automatic verification

1 Introduction

Security requirements of a computer system are regarded as critical properties that demand the availability of mechanisms which control or protect programs and data. Three issues in particular are related to the area of computer security: (i) *confidentiality* (*secrecy*), the problem of protecting information from unauthorised disclosure; (ii) *integrity*, the protection of information from unauthorised modification or destruction; and (iii) *denial of service*, the avoidance of major reduction in system performance.

It is possible to regard these security concerns as properties of *information flow* within the system and base a specification of security on the absence of undesired flows. The notion of *non-interference* captures the idea that no information can flow from one user to another if the system view of the second is completely unaffected by actions of the first. We introduce a novel characterisation of non-interference based on the notion of deterministic views. This elegant formulation of non-interference has, unlike others described in the literature, the property of preserving security requirements under refinement.

The development of a secure system entails the construction of an abstract security model in addition to the specification of the system's functional requirements. The model is intended to capture abstractly the complete set of security requirements, which are derived from the system's (possibly informal) security

policy, and which form part of the total system requirements. Depending on the level of rigour required during development, it is necessary to either informally establish or formally prove a *correspondence* between the functional specification and the abstract model.

The construction of the security model has been attempted [Col94, Jon92] with the same methods as functional specifications, such as the Z notation [Spi92]. We suggest that there are good reasons to employ a process algebraic notation for this purpose. Firstly, it is not the individual operations of the system, but the system as a whole that is to satisfy critical properties. Secondly, insecurity is introduced not by a single operation in isolation but by certain sequences of operations. And thirdly, it turns out to be possible to express non-interference constraints directly on a process representation of the system, thus eliminating the need for constructing a separate abstract model.

We therefore propose a process-algebraic approach (based on CSP [Hoa85]) to the specification of security properties. In particular the property of a process being *deterministic* is fundamental to the conditions we introduce for non-interference. This property can be verified using standard algorithms on finite-state systems, such as those implemented in the CSP model checker FDR[1] [Ros94a].

This paper is organised as follows. The following section defines the non-interference conditions and illustrates some of their properties. The conditions are generalised to systems with multiple users. Section 3 presents a functional specification of a file systems that is intended to maintain confidential information. A systematic way of mapping this specification into process algebra is given in section 4, and the particular conditions for non-interference in the process model are clearly stated. The security flaw of the system is detected by automatic verification in section 5, and it is shown how the system can be made secure. Finally, we present our conclusions in section 6.

2 Non-interference and Determinism

There have been a number of CSP formulations of non-interference, such as Jacob's use of inference functions [Jac90]. None of these approaches is based on the notion of determinism, which has only recently been recognised as the fundamental concept underlying the various definitions of non-interference [Ros94b]. This section will introduce some formal definitions of non-interference and analyse their properties.

2.1 Notation and Conventions

We will employ the failures-divergences model of CSP, in which a process is characterised by its failures and its divergences. We use the following notation to refer to various observations of a process P.

[1] FDR (Failures Divergence Refinement) is a product of Formal Systems (Europe) Ltd., 3 Alfred St., Oxford OX1 3EH, UK.

$\alpha(P)$	alphabet	set of events process P can engage in
TRACES(P)	traces	set of finite sequences of events P can engage in
FAILS(P)	failures	set of pairs (s, X) such that P can refuse events X after trace s
DIVS(P)	divergences	set of traces after which P may behave chaotically

The semantics of the *failures-divergences model* of CSP is detailed in (e. g.) [Hoa85]. Of particular relevance below will be the concealment and interleaving operators whose formal semantics are given in the Appendix. Informally, $P \setminus A$ is a process that behaves like P except that occurrences of events in set A are concealed. A concealed event occurs automatically and instantaneously as soon as it can, without being observed or controlled by the environment of the process. The $|||$ interleaving operator models asynchronous composition of processes: $P \;|||\; Q$ is a process whose trace forms an arbitrary interleaving of events from processes P and Q. An event can be refused by the composition only if both component processes refuse it.

We will interpret some processes U_i as users interacting with another process P called the system. A user U of P is defined by its interface to the system. For the moment, it is assumed that the system has only two users U_H and U_L, with $\alpha(P) = H \cup L$ and $H \cap L = \varnothing$. This latter condition of disjoint set of actions available to the users is convenient since it prohibits direct communication between users by synchronisation.

These simplifying assumptions will be relaxed in section 2.5 where the non-interference conditions will be generalised to multi-user systems.

2.2 Abstracting Events

In a system with two users U_H and U_L we will typically want one user (U_L) to be completely unaware of what the other (U_H) does. In other words, the system view of U_L should be unaffected by the presence or absence of events user U_H might engage in. If this is the case, we say that there is no flow of information from U_H to U_L, or that U_H is non-interfering with U_L.

In a sense, it is necessary to abstract away from the actual or potential behaviour of U_H and ensure that this abstraction cannot affect how the system appears to U_L. There are several ways this abstraction may be captured, e.g. by concealing or obscuring U_H's actions. In CSP, the concealment of events is expressed using the $\setminus$ hiding operator, and the obscuring of events may be achieved using $|||$ interleaving.

It is well-known that concealment and interleaving of events may introduce non-determinism [Hoa85, pp. 113 and 120]. A non-deterministic system may, under the same conditions, behave differently towards its environment, due to some internal, uncontrollable choice. Though this choice cannot be observed directly, its external effects can, and thus provide clues on abstracted activities. The result of this abstraction will be that U_H's actions become choices which, though not visibly directly to U_L, may resolve non-determinism that is. The absence of non-determinism under abstraction of U_H's behaviour guarantees the absence of undesired information flow towards U_L.

The notion of determinism is formally defined as follows. A process P is *deterministic* if it is free of divergence, and if it never has a choice whether to refuse an event it can engage in.

$$P \textbf{ det } \Leftrightarrow \textsc{Divs}(P) = \varnothing \wedge (tr^\frown\langle a\rangle \in \textsc{Traces}(P) \Rightarrow (tr, \{a\}) \notin \textsc{Fails}(P))$$

A process lacking this property is *non-deterministic*; under identical environmental conditions it may behave differently in an unpredictable fashion.

2.3 Non-interference Conditions

The conditions we propose are all based on the absence of non-determinism after the abstraction of "high-security" events, and are justified in detail in [Ros94b]. Concealment is the simplest method of abstracting from events in CSP which can serve as a first attempt to define the notion of non-interference.

Definition 1. A system P is said to be *eagerly secure* with respect to H if concealment of H events does not introduce non-determinism, i.e.

$$\textbf{E-Sec}_H(P) \Leftrightarrow (P \setminus H) \textbf{ det}$$

The terminology will become clear later on. Another way of abstracting events in CSP is not by concealing but by obscuring their occurrence. This can be achieved using another process

$$RUN_H = x : H \rightarrow RUN_H$$

and combining it with the original system P by interleaving as $P \,|||\, RUN_H$.

This process can never refuse an H event since RUN_H is always prepared to contribute one in arbitrary places. An outside observer will not be able to tell whether such an action came originally from P or from RUN_H. As above, we postulate that abstraction by interleaving does not introduce non-determinism.

Definition 2. A system P is said to be *lazily secure* with respect to H if obscuring H events by interleaving does not introduce non-determinism, i.e.

$$\textbf{L-Sec}_H(P) \Leftrightarrow (P \,|||\, RUN_H) \textbf{ det}$$

Example 1. Consider the system P with $H = \{h_1, h_2\}$ and $L = \{l\}$ defined

$$P = (h_1 \rightarrow l \rightarrow P) \,\Box\, (h_2 \rightarrow l \rightarrow P)$$

The system repeatedly offers a choice of a single H event followed by action l. Concealing H permits U_L to engage in l whenever desired independently of the (hidden) choice between h_1 and h_2. Hence U_L's view of $P \setminus H$ is deterministic and $\textbf{E-Sec}_H(P)$ holds. We may doubt, however, whether the system should really be regarded as secure because the availability of l depends on the previous occurrence of an H action. The lazy condition does not make the assumption that H actions occur so quickly such that no refusal to communicate l may be recognised by U_L. System P therefore fails to be lazily secure, reflecting a dependence of U_L's system view on activity of the other user. □

The terminology of the conditions reflects the semantics of the operators involved. The \ concealment operator is defined in a way such that hidden (internal) actions occur instantly. Abstraction of events by concealment is eager in the sense that the events cannot be prevented or delayed by the environment.

This situation contrasts with the usual interpretation of communications between interacting processes. The standard interpretation of the occurrence of an event is that the process and its environment have agreed on the action; it cannot occur without mutual consent. The agreement of U_H to engage in H events cannot be assumed to be immediately forthcoming. Abstraction by interleaving $P \mid\mid\mid RUN_H$ does not force events from P to happen, it simply prevents an observer from knowing whether they came from P or from RUN_H. This lack of urgency explains why this is lazy abstraction. $P \mid\mid\mid RUN_H$ can only be deterministic if the set of L events available before and after any H event of P are the same, since if the same event is communicated by RUN_H the state of P does not change. Lazy abstraction is thus sensitive not only to the effects of different actions by U_H, but also to the choice between action and inaction.

The possibility of infinite sequences of H actions give rise to the danger that a system implementation will prefer them forever, thereby denying U_L the opportunity to communicate—which would be a clear breach of security. The eager security condition, which entails the assumption that H actions are never delayed, is necessarily sensitive to this possibility as $P \setminus H$ introduces divergence.[2]

Example 2. Let $H = \{d_1, d_2, s_1, s_2\}$ and $L = \{l_1, l_2\}$. In the system

$$\begin{aligned} Q = &(l_1 \rightarrow l_2 \rightarrow P) \\ &\Box\, (d_1 \rightarrow s_1 \rightarrow P) \\ &\Box\, (d_2 \rightarrow s_2 \rightarrow P) \end{aligned}$$

there is the possibility of an infinite sequence of H actions. This potentially endless delay of U_L's request is flagged by the eager condition since $Q \setminus H$ diverges, so Q is not eagerly secure. The system also fails the lazy condition since event l_1 will be removed from the interface when U_H engages in either d_1 or d_2. Thus U_H will delay the system by communicating d_1 or d_2 until a further s action is taken. User U_L will recognise that the system refuses a request l_1 before U_H's request is complete. □

Whether the lack of lazy security in Example 2 should be regarded as a security breach depends on the nature of events $\{s_1, s_2\}$. If these are events which occur instantaneously—such as a system message appearing on the user's screen— then they are indistinguishable to U_L from the ordinary internal actions of Q. As long as these "signal" events are guaranteed to occur instantaneously there will be no refusal of a request by U_L at the interface to the system.

The H events can therefore be divided into two categories: signal events S which are guaranteed to occur instantly, and events D which cannot occur without the agreement of U_H and may thus be delayed. In many systems, delayable

[2]The lazy condition (where H actions may be subject to delay) assumes that the implementation is sufficiently fair to avoid this insecurity.

events take the form of inputs whereas the signals appear as output communications to the environment (including users).

Since S events resemble internal system actions we can abstract from them by hiding while we still use interleaving for ordinary events such as $\{d_1, d_2\}$ above. The combination of the two forms of abstraction results in a *mixed* non-interference condition.

Definition 3. A system P whose H events can be partitioned into delay events D and signal events S satisfies $\textbf{M-Sec}_{(D,S)}(P)$ if $(P \setminus S) \,|||\, RUN_D$ is deterministic.

2.4 Properties of Conditions

From the eager and lazy conditions based on the notion of determinism it is possible to derive conditions involving only observations of the process concerned. Eager security can be paraphrased as stating that nothing which is observed in L after trace tr will allow the H events which happened during tr to be inferred.

Proposition 4. *If system P satisfies $\textbf{E-Sec}_H(P)$, then $P \setminus H$ is free of divergence, and for any two traces $tr, tr' \in \text{TRACES}(P)$,*

$$tr \restriction L = tr' \restriction L \;\Rightarrow\; (P/tr) \setminus H =_{FD} (P/tr') \setminus H$$

A corresponding consequence can be derived from the definition of lazy security.

Proposition 5. *If system P satisfies $\textbf{L-Sec}_H(P)$, then P is free of divergence, and for any two traces $tr, tr' \in \text{TRACES}(P)$,*

$$tr \restriction L = tr' \restriction L \;\Rightarrow\; (P/tr) \,|||\, RUN_H =_{FD} (P/tr') \,|||\, RUN_H$$

The approach of postulating determinism after abstraction of high-security events can be generalised by analysing various models of U_H. The framework in which this can be done is provided by the condition

$$(P \,[\![H]\!]\, U) \setminus H \;\; \textbf{det} \tag{1}$$

for a suitably chosen process U which has to synchronise with P on every event in H. Process U can be regarded as a model of user U_H.

The user with the widest range of behaviour is one whose actions are unpredictable and uncontrollable. Such activity is represented in CSP by a process $CHAOS$ defined as

$$CHAOS_H = STOP \sqcap (x : H \rightarrow CHAOS_H)$$

displaying the most non-deterministic behaviour which is free of divergence. A system with such a non-deterministic user will lack interference only in the case of both eager and lazy security.

Proposition 6. *A system P satisfies $\textbf{E-Sec}_H(P)$ and $\textbf{L-Sec}_H(P)$ if, and only if, the process $(P \,[\![H]\!]\, CHAOS_H) \setminus H$ is deterministic.*

It is shown in [Ros94b] that all three non-interference conditions (eager, lazy, and mixed) can in fact be expressed in the form of (1). For eager security, the model for user U_H is simply identical to RUN_H since $P \,|[\, H \,]|\, RUN_H = P$ for all processes P.

Corresponding formulations for lazy and mixed security require a more powerful model of CSP. In the infinite traces model [Ros93] the failures-divergences representation of a process is augmented with its set of infinite traces. The model of user U_H required for lazy security is a process $FINITE_H$ which behaves just like $CHAOS_H$ but without ever engaging in an infinite trace. This restriction prohibits the occurrence of infinite H sequences resulting in divergence under concealment.

Proposition 7. *Eager, lazy, and mixed security can be all be expressed in the general form of (1) as follows.*

$$\begin{aligned}
\textbf{E-Sec}_H(P) &\Leftrightarrow (P \,|[\, H \,]|\, RUN_H) \setminus H \textbf{ det} \\
\textbf{L-Sec}_H(P) &\Leftrightarrow (P \,|[\, H \,]|\, FINITE_H) \setminus H \textbf{ det} \\
\textbf{M-Sec}_{(D,S)}(P) &\Leftrightarrow (P \,|[\, H \,]|\, (RUN_S \,|||\, FINITE_D)) \setminus H \textbf{ det}
\end{aligned}$$

These various 'users' suggest a more general approach to security specification: for a particular context, choose a process U which characterises all possible behaviours of U_H under which it is expected that confidentiality will be maintained. Usually this will be all its behaviours, but it is possible to imagine other circumstances, for example if the system P represents a mail system where it is allowable for a high-security user to send a message to a low-security one, we might expect to maintain confidentiality so long as no such messages are sent. (This type of property is known as *conditional* non-interference.)

The more non-deterministic the abstract model U the stronger is the equivalent security condition. When a more deterministic process is substituted for U, the properties of CSP refinement guarantee the preservation of non-interference.

More precisely, if P is a system component in context C, then refinement of P – replacing it with a less non-deterministic component – preserves determinism of the original system:

$$C(P) \textbf{ det} \;\wedge\; P \sqsubseteq P' \;\Rightarrow\; C(P') \textbf{ det}$$

It is equally a consequence of this fact that refining P preserves the determinism of $(P \,|[\, H \,]|\, U) \setminus H$, and that therefore each of our non-interference properties is preserved under refinement. This is a result which may be exploited in system development or maintenance.

Proposition 8. *Eager, lazy, and mixed security are preserved under refinement:*

$$\begin{aligned}
\textbf{E-Sec}_H(C(P)) \wedge P \sqsubseteq P' &\Rightarrow \textbf{E-Sec}_H(C(P')) \\
\textbf{L-Sec}_H(C(P)) \wedge P \sqsubseteq P' &\Rightarrow \textbf{L-Sec}_H(C(P')) \\
\textbf{M-Sec}_{(D,S)}(C(P)) \wedge P \sqsubseteq P' &\Rightarrow \textbf{M-Sec}_{(D,S)}(C(P'))
\end{aligned}$$

A number of additional results concerning the composition and decomposition of secure systems may be derived; see [Ros94b]. One such result is that a system P may be decomposed into two non-interacting parts if it is lazily secure with respect to two disjoint alphabets.

Proposition 9. *Let A, B be disjoint alphabets.* **L-Sec**$_A(P)$ *and* **L-Sec**$_B(P)$ *hold of a system if, and only if, there are two deterministic processes P_A with $\alpha(P_A) = A$ and P_B with $\alpha(P_B) = B$ such that $P = P_A \parallel P_B$.*

Further properties of our non-interference conditions as well as the proofs of the propositions in this section may be found in [Ros94b].

2.5 Generalisation

We will now generalise the determinism conditions for multi-user systems. If F is the system whose non-interference properties we attempt to establish, the system model can be described as

$$SYSTEM = Users \parallel F \quad \text{where} \quad Users = |||_{i>0}\, U_i$$

It is assumed that there is a security classification associated with each user. Let $CLASS$ be the partially ordered set of these classifications. The total function $cl : Users \rightarrow CLASS$ assigns one classification to each user process. A further assumption is $\alpha(U_i) \cap \alpha(U_j) = \varnothing$ whenever $cl(U_i) \neq cl(U_j)$.

The function $above : CLASS \rightarrow \mathbb{P}\alpha(SYSTEM)$ is used to define the set of events that should be hidden from a user operating on a particular level of classification, which is given by

$$above(c) = \alpha(SYSTEM) - above^{-1}(c)$$

where

$$above^{-1}(c) = \bigcup\{\alpha(U_i) \mid cl(U_i) \leq c\}$$

The non-interference conditions of section 2.3 hold for a multi-user system if they hold on each security level of the system.

Definition 10. A multi-user system P is eagerly and (respectively) lazily-secure if,

$$\forall\, c_i \in CLASS \bullet \textbf{E-Sec}_{H_j}(P)\ , \text{ and}$$
$$\forall\, c_i \in CLASS \bullet \textbf{L-Sec}_{H_j}(P)$$

where $H_j = above(c_i)$.

In a realistic system it is typically the mixed non-interference condition that requires verification on each security level, as will be illustrated in the following case study.

3 A "Secure" File System

This and the following section will illustrate the framework in which our non-interference conditions can be applied. The example is that of a file system in which confidential data is to be maintained.

It is widely accepted that a formal specification can increase the level of assurance that a system will meet its security requirements [Gas88]. In fact governmental standards for the development of secure systems mandate the use of formal methods and proof. The Z notation [Spi92] is particularly suited for this task since (i) it has a well-defined semantics; (ii) it has been successfully employed in industrial scale software development; and (iii) it has become increasingly popular for the specification and verification of secure systems [Col94, Jon92].

We begin the specification of the file system by introducing some basic types. The set of users of the system is represented by type *USER*, each of which holds an associated security classification from the set *CLASS*. *FID* represents the set of file identifiers, and *DATA* refers to the set of possible data that may be stored in a file. This type contains a special value *NULL* representing invalid data. These are the basic types we will use

$$[USER, CLASS, FID, DATA]$$

There is a security classification associated with each user. We use a global function *cl* to obtain the appropriate class by supplying it with a user identification. It is declared as a *total* function; there cannot be users without classification.

$$cl : USER \rightarrow CLASS$$

3.1 File Model

Each file has the structure[3]

File

$$class : CLASS$$
$$data : DATA$$

where the *class* component relates to the level of security of the stored *data*. Each file initialised with the level of clearance at which the file is created.

Init

$$File'$$
$$clear? : CLASS$$

$$class' = clear?$$
$$data' = NULL$$

[3] In Z, formal notation is separated from informal descriptions by so-called schema boxes. A schema contains a number of declarations and, if there are any constraints on these declarations, a separating line followed by appropriate predicates. Schemas are used to represent structured state as well as operations on structures.

We follow a standard convention of decorating inputs with ?, outputs with !, and states after completion of the operation with a prime ′. Unprimed variables or schemas refer to states before the operation.

Two operations are provided on files: reading stored data, and writing new data, provided the file access is carried out with the appropriate clearance. Reading is permitted only when the operation is carried out with appropriate access permission $clear? \geq class$, in which case stored data is output as $data!$. The notation $\Xi File$ indicates that reading a file does not change its state.

Rd0

$$\Xi File$$
$$clear? : CLASS$$
$$data! = data$$
$$\hline$$
$$clear? \geq class$$
$$data! = data$$

Storing new data in a file is carried out with a *Wr0* operation which is permitted only if the user clearance is equal to the file classification. The $\Delta File$ schema component indicates that writing data changes the file state; the input data $new?$ is stored in the *data* component of *File*.

Wr0

$$\Delta File$$
$$clear? : CLASS$$
$$new? : DATA$$
$$\hline$$
$$clear? = class = class'$$
$$data' = new?$$

To indicate the success or failure of an operation, we define the system's response as type

$$RESP ::= ok \mid fail$$

Each operation on a file is accompanied by an indication of whether it has succeeded. The output message is defined by the (horizontal) schemas

$$Success \mathrel{\widehat=} [resp! : RESP \mid resp! = ok]$$
$$Failure \mathrel{\widehat=} [resp! : RESP \mid resp! = fail]$$

We do not give the user any indication of whether a failure was caused by a functional error or a security breach, in order to avoid a potential channel of information flow.

If a request for file access is carried out without valid clearance the operation fails, and the file status remains unchanged ($\Xi File$). The case of invalid read access is described as

NoRdAccess

$\Xi File$
Failure
$clear? : CLASS$
$data! : DATA$

$clear? < class$
$data! = NULL$

The corresponding error condition for writing is

NoWrAccess

$\Xi File$
Failure
$clear? : CLASS$

$clear? \neq class$

The total read and write operations are Rd and Wr specified as

$$Rd \;\widehat{=}\; (Rd0 \wedge Success) \vee NoRdAccess$$
$$Wr \;\widehat{=}\; (Wr0 \wedge Success) \vee NoWrAccess$$

If the request is carried out with appropriate clearance the system reports with *ok*, otherwise the user just receives a *fail* message and the file remains unaltered.

3.2 File System

Our file system is given by

FileSystem

$files : FID \pfun File$

Component *files* is declared as a partial function from file identifiers to files. This means that no two files can have the same name. The system initially contains no files:

$$FInit \;\widehat{=}\; [FileSystem' \mid files' = \varnothing]$$

In addition to the initialisation occurring when a file is created at the system level, we want the operations of reading and writing a file to be available at the system interface. This is a achieved by *promoting* the schemas *Init*, *Rd*, and *Wr* with the aid of two "framing" schemas:

$\Phi 1$

$\Delta FileSystem$
$file? : FID$
$user? : USER$

$clear? = cl(user?)$
$files' = files \oplus \{file? \mapsto \theta File'\}$

$\Phi 2$

$\Phi 1$

$file? \in \mathrm{dom}\, files$

$\theta File = files(file?)$

The promoted operations will require both a file name (*file*?) and a user identification (*user*?) as input. The user's classification is then the clearance at which the file operation is carried out. The three operations available at the interface are

$$Create0 \;\widehat{=}\; \exists File' \bullet (\Phi 1 \wedge Init)$$
$$Read0 \;\widehat{=}\; \exists \Delta File \bullet (\Phi 2 \wedge Rd)$$
$$Write0 \;\widehat{=}\; \exists \Delta File \bullet (\Phi 2 \wedge Wr)$$

It is necessary to ensure that no operation is carried out on files which do not exist. This error condition can occur if the user supplies an invalid file identifier.

UnknownFile

$\Xi FileSystem$

$Failure$

$file? : FID$

$file? \notin \mathrm{dom}\, files$

Similarly, a request for file creation cannot succeed if the suggested name has already been used for another file.

FileExists

$\Xi FileSystem$

$Failure$

$file? : FID$

$file? \in \mathrm{dom}\, files$

The total operations available at the file system interface are then given by

$$Create \;\widehat{=}\; (Create0 \wedge Success) \vee FileExists$$
$$Read \;\widehat{=}\; Read0 \vee UnknownFile$$
$$Write \;\widehat{=}\; Write0 \vee UnknownFile$$

We suggest that a security analysis is best carried out on a process algebraic representation of the system. This representation may be regarded as a security model [Gas88] which can in fact be derived by translation. It is therefore unnecessary to engage in an error-prone attempt to prove a correspondence between model and specification. In the coming section we map the functional specification of the file system into CSP and state the non-interference conditions that require verification.

4 Z into CSP

The Z specification may be translated into CSP according to the technique described in [Woo94]. The theoretical basis for this work may be found in [WM90].

First we interpret the Z specification as an action system [BKS83] whose state is specified by *File*. It has two actions corresponding to the operations *Rd* and *Wr*. However, each of these operations also has an output, and we must be careful to separate the two parts of the operation and associate an action with each, since we cannot regard input and output as happening simultaneously. When a user has invoked an operation, but has not consumed its output, then the system will do nothing else while that output is pending. When no output is pending, all operation actions are enabled.

This interpretation of a Z specification is informal (albeit systematic), but it does correspond to the intuitive meaning given to Z specifications (see [Spi92], for example).

Consider the *Wr* operation. We must separate it into two parts: the first part consumes the input and then stores its output in the state; the second part waits for the opportunity of delivering its output. Define a new free type that is either a response or nothing:

$$RESP_+ ::= nullresp \mid outresp\langle\langle RESP\rangle\rangle$$

and augment the state of a file with a component that contains the pending output (if it exists)

$File_+$

$File$

$wrpend : RESP_+$

The first part of the operation is as follows

Wr_+

$\Delta File_+$

$clear? : CLASS$

$new? : DATA$

$wrpend = nullresp$

$\exists\, resp! : RESP \mid wrpend' = outresp(resp!) \bullet Wr$

and the second part is

$Wr\_$

$\Delta File_+$

$\Xi File$

$resp! : RESP$

$wrpend \neq nullresp$

$resp! = outresp^{\sim}(wrpend)$

$wrpend' = nullresp$

We can prove that the only change we are making to Wr by splitting into two is to delay its output:

$$\vdash Wr = \exists\, wrpend, wrpend' : RESP_+ \bullet Wr_+ \fatsemi Wr_-$$

According to [Woo94], we can now translate our specification of the write operation into two actions.

$$wr?clear?new \wedge wrpend = nullresp \rightarrow Wr_+$$

$$wrout!(outresp^\sim(wrpend)) \wedge wrpend \neq nullresp \rightarrow wrpend := nullresp$$

Thus, upon receipt of the communication of a clearance and some new data, then, providing that there is no write-output pending, the Wr_+ operation is performed. Output may be transmitted whenever it is pending.

We can make similar transformations for the other operations.

The actions may now be embedded in a CSP-framework process. We now have a CSP process which is formally equivalent to the *File* abstract data type.

$$File = init?class \rightarrow File(class, NULL, (nullresp, nulldata), nullresp)$$

$$\begin{array}{l}
File(class, data, rdpend, wrpend) = \\
\quad \textbf{if } rdpend = nullresp \wedge wrpend = nullresp \\
\quad \textbf{then} \\
\qquad rd?clear \rightarrow \\
\qquad\quad \textbf{if } clear \geq class \\
\qquad\quad \textbf{then } File(class, data, (outresp(ok), outdata(data)), wrpend) \\
\qquad\quad \textbf{else } File(class, data, (outresp(fail), outdata(NULL)), wrpend) \\
\qquad \Box\ wr?clear?new \rightarrow \\
\qquad\quad \textbf{if } clear = class \\
\qquad\quad \textbf{then } File(class, new, rdpend, outresp(ok)) \\
\qquad\quad \textbf{else } File(class, data, rdpend, outresp(fail)) \\
\quad \textbf{else} \\
\qquad \textbf{if } rdpend \neq nullresp \\
\qquad \textbf{then } rdout!outresp^\sim(rdpend) \\
\qquad\quad \rightarrow File(class, data, (nullresp, nulldata), wrpend) \\
\qquad \textbf{else } wrout!outresp^\sim(wrpend) \\
\qquad\quad \rightarrow File(class, data, rdpend, nullresp)
\end{array}$$

The *File* process may now be transformed using the laws of CSP, and, if desired, the state variables containing the pending outputs elided.

In [Woo94], the connection is made between the technique of promotion in Z, and the use of subordination or the means of sharing through interleaving. In this way, the file system can be created as a system of CSP processes.

The structure of the resulting CSP implementation is illustrated in Figure 1. *FILES* will be a shared pool of files accessible through the interface *FSYS*. Each file has an associated name and classification, and may contain arbitrary data. The process *File* models a file waiting to be initialised.

$$File = init?file?class \rightarrow File(file, class, NULL)$$

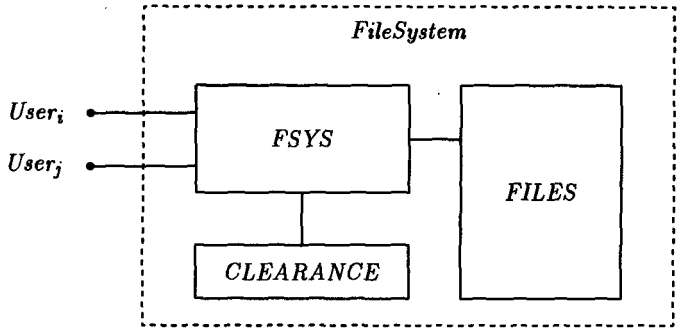

Fig. 1. The file system implemented by communicating processes.

A file after initialisation may be read or written to.

$$File(file, class, data) = Rd(file, class, data) \,\Box\, Wr(file, class, data)$$

The read operation is implemented by process Rd as

$$\begin{aligned} &Rd(file, class, data) = rd.file?clear \rightarrow \\ &\quad \textbf{if } clear \geq class \\ &\quad \textbf{then } rdout.file!ok!data \rightarrow File(file, class, data) \\ &\quad \textbf{else } rdout.file!fail!NULL \rightarrow File(file, class, data) \end{aligned}$$

Storing new data in a file is realised with process

$$\begin{aligned} &Wr(file, class, data) = wr.file?clear?new \rightarrow \\ &\quad \textbf{if } clear = class \\ &\quad \textbf{then } wrout.file!ok \rightarrow File(file, class, new) \\ &\quad \textbf{else } wrout.file!fail \rightarrow File(file, class, data) \end{aligned}$$

The total pool of files is given by

$$FILES = |||_{0 \leq i < n} File$$

4.1 The System Interface

It is not possible to conjoin $FILES$ with the set of user processes directly because users must be protected from a number of functional errors, such as reading a file which does not exist. To this purpose, we will provide a system interface process $FSYS$ which manages access to the individual files.

$$\begin{aligned} \alpha(FSYS) = \{&create, createout, read, readout, write, writeout, \\ &init, rd, rdout, wr, wrout, clear\} \end{aligned}$$

$FSYS$ holds state variable *files*, the set of current file names

$$FSYS(files) = Create(files) \,\Box\, Read(files) \,\Box\, Write(files)$$

The three services available at the interface are implemented with processes *Create*, *Read*, and *Write* respectively.

$$
\begin{aligned}
&\mathit{Create}(\mathit{files}) = \mathit{create}?\mathit{user}?\mathit{file} \rightarrow \\
&\quad \textbf{if } \mathit{file} \notin \mathit{files} \\
&\quad \textbf{then } \mathit{clear.user}?\mathit{class} \rightarrow \mathit{init}!\mathit{file}!\mathit{class} \rightarrow \mathit{createout.user}!\mathit{ok} \\
&\qquad \rightarrow \mathit{FSYS}(\mathit{files} \cup \{\mathit{file}\}) \\
&\quad \textbf{else } \mathit{createout.user}!\mathit{fail} \rightarrow \mathit{FSYS}(\mathit{files})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Read}(\mathit{files}) = \mathit{read}?\mathit{user}?\mathit{file} \rightarrow \\
&\quad \textbf{if } \mathit{file} \in \mathit{files} \\
&\quad \textbf{then } \mathit{clear.user}?\mathit{class} \rightarrow \mathit{rd.file}!\mathit{class} \rightarrow \mathit{rdout.file}?\mathit{resp}?\mathit{result} \\
&\qquad \rightarrow \mathit{readout.user.file}!\mathit{resp}!\mathit{result} \rightarrow \mathit{FSYS}(\mathit{files}) \\
&\quad \textbf{else } \mathit{readout.user.file}!\mathit{fail}!\mathit{NULL} \rightarrow \mathit{FSYS}(\mathit{files})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Write}(\mathit{files}) = \mathit{write}?\mathit{user}?\mathit{file}?\mathit{new} \rightarrow \\
&\quad \textbf{if } \mathit{file} \in \mathit{files} \\
&\quad \textbf{then } \mathit{clear.user}?\mathit{class} \rightarrow \mathit{wr.file}!\mathit{class}!\mathit{new} \rightarrow \mathit{wrout.file}?\mathit{resp} \\
&\qquad \rightarrow \mathit{writeout.user.file}!\mathit{resp} \rightarrow \mathit{FSYS}(\mathit{files}) \\
&\quad \textbf{else } \mathit{writeout.user.file}!\mathit{fail} \rightarrow \mathit{FSYS}(\mathit{files})
\end{aligned}
$$

Process *CLEARANCE* provides the appropriate classification of a user when required.

$$\mathit{CLEARANCE} = (\Box\ \mathit{clear.u}!(\mathit{cl}(u)) \rightarrow \mathit{CLEARANCE}) \quad \text{for all } u \in \mathit{USER}$$

The complete file system is given by parallel composition of the interface process, the file pool, and the clearance process, with intermediate channels concealed.

$$
\begin{aligned}
\mathit{FileSystem} = \ & (\mathit{FSYS}(\varnothing) \parallel \mathit{FILES} \parallel \mathit{CLEARANCE}) \\
& \setminus \{\mathit{init}, \mathit{rd}, \mathit{rdout}, \mathit{wr}, \mathit{wrout}, \mathit{clear}\}
\end{aligned}
$$

4.2 Security Specification

Any particular instance of the file system can be subjected to the security conditions presented in section 2. We will consider the case of three users with the following classifications.

USER	*CLASS*
Lisa	3 (highest)
Mari	2
Nina	1 (lowest)

It is convenient to partition the events at the system interface into "delay" and "signal" events on each level of user classification (except the top level).

$$
\begin{aligned}
\mathit{H2d} &= \{\ \mathit{create.user.file}, \mathit{read.user.file}, \mathit{write.user.file.data} \mid \mathit{user} \in \{\mathit{Lisa}\}\ \} \\
\mathit{H2s} &= \{\ \mathit{createout.user.resp}, \mathit{readout.user.file.resp.data}, \\
&\qquad \mathit{writeout.user.file.resp} \mid \mathit{user} \in \{\mathit{Lisa}\}\ \}
\end{aligned}
$$

$$H1d = \{\, create.user.file, read.user.file, write.user.file.data \mid user \in \{Lisa, Mari\} \,\}$$
$$H1s = \{\, createout.user.resp, readout.user.file.resp.data, writeout.user.file.resp \mid user \in \{Lisa, Mari\} \,\}$$

The file system satisfies **E-Sec**($FileSystem$) if

$$(FileSystem \setminus (H2d \cup H2s))\ \mathbf{det}\ \wedge (FileSystem \setminus (H1d \cup H1s))\ \mathbf{det}$$

The file system is lazily secure if

$$(FileSystem \,|||\, RUN_{(H2d \cup H2s)})\ \mathbf{det}\ \wedge (FileSystem \,|||\, RUN_{(H1d \cup H1s)})\ \mathbf{det}$$

The file system satisfies the mixed security property if

$$((FileSystem \setminus H2s) \,|||\, RUN_{H2d})\ \mathbf{det}\ \wedge$$
$$((FileSystem \setminus H1s) \,|||\, RUN_{H1d})\ \mathbf{det}$$

It turns out that none of these conditions is met – i. e. that the system contains undesired information flows. Since it may not be obvious that the conditions fail to hold (and why not), we employ a verification tool.

5 Automated Verification

The effort of formulating the eager/lazy/mixed non-interference conditions would be futile without a method of verifying them. Luckily, the absence of non-determinism on which the conditions are based can be automatically verified using standard algorithms on finite-state systems. We show that the CSP proof tool FDR can be used to complete the security analysis.

5.1 FDR

The FDR tool [Ros94a] has been originally designed to verify behavioural CSP specifications, in particular refinement relations between processes. These refinement checking capabilities are employed to decide whether a given process is deterministic using the following algorithm:

1. Search through the state space of P, resolving all non-determinism that is encountered. In a "stable" state (in which internal progress is impossible) a single representative for each available action is selected, whereas in a state where internal actions are possible we chose one of these arbitrarily. This search either finds a divergence of P (in which case it is clearly non-deterministic) or yields a deterministic process Q that refines the original P.
2. Use the refinement checker to confirm whether $Q \sqsubseteq P$. The check succeeds if, and only if, P is deterministic.

The algorithm is justified by the fact that the deterministic processes are maximal in the failures-divergences model of CSP, and are therefore incomparable. Thus, for some arbitrary deterministic refinement Q of P,

$$P\ \mathbf{det} \Leftrightarrow P =_{FD} Q$$

5.2 Making the File System Secure

Checking the security specification of section 4.2 using FDR confirms that the file system is neither eagerly nor lazily secure. The reasons for this lie in the basic structure of the system interface: a menu of services is offered to users with various classifications, and a choice of service by a particular user is followed by a system response on the same security level.

This structure resembles that of the (much simpler) process Q of Example 2 which was already observed to be insecure under the eager and lazy conditions. As was motivated there, these conditions are inappropriate for a system structured like Q or *FileSystem*, and it becomes necessary to partition events into delay and signal events in order to apply the mixed condition.

However **M-Sec**(*FileSystem*) fails to hold as well, which must be of serious concern to the system designers. A check using FDR shows the reason for this to be the possible failure of a request to create a file. The file system was specified to prohibit the existence of two files with the same name. This feature is a security flaw since a user who attempts to create a file (with identifier *id* say) and fails has learned that a file named *id* of higher classification exists. This clear breach of non-interference is reflected in the failure of the mixed condition.

The question remains how the flaw can be overcome. One idea may be to change the *Create* operation so that a request of file creation always succeeds. This approach is probably unsatisfactory if creation of a file which already exists results in stored data to be lost. A more promising approach is to somehow associate classifications with file identifiers in order to guarantee that files on different security levels have different names.

A simple way of implementing this is to provide pairwise disjoint sets of identifiers for the different levels. For the system in section 4 one might consider partitioning the set *FID* into three sets (say)

$$FID_1 = \{a, b\},\ FID_2 = \{c, d\},\ FID_3 = \{e, f\}$$

so that for all $i \in CLASS$

$$FID = \bigcup FID_i$$

Doing so entails the re-definition of the *Create* operation which now needs to confirm whether the use of a particular identifier is valid with regard to the user's classification:

$$\begin{array}{l} Create(files) = create?user?file \rightarrow \\ \quad \textbf{if } file \notin files \\ \quad \textbf{then } clear.user?class \rightarrow \\ \quad\quad \textbf{if } file \in FID_{class} \\ \quad\quad \textbf{then } init!file!class \rightarrow createout.user!ok \rightarrow FSYS(files \cup \{file\}) \\ \quad\quad \textbf{else } createout.user!fail \rightarrow FSYS(files) \\ \quad \textbf{else } createout.user!fail \rightarrow FSYS(files) \end{array}$$

Verification of the mixed security condition now shows that **M-Sec**(*FileSystem*) does indeed hold, *provided* that the number of files available through the system is

equal to or exceeds the combined total of identifiers for all levels of classification. So if

$$FILES = |||_{0 \leq i < n} File$$

we require $n \geq size(FID)$. Without the proviso the file system does not pass the mixed condition, again because an attempt of file creation may fail. This time the security breach is caused by the potential exhaustion of the pool of available files.

6 Conclusion

This paper presents process algebraic specifications as a practical framework for the development of systems with security constraints. The approach is illustrated with an example of a file system intended to maintain secret data, but in fact our results apply equally to systems with security concerns other than confidentiality. This is a consequence of defining general non-interference conditions which require the system view of particular users to be unaffected by the actions taken by others.

Our non-interference conditions are based on the notion of deterministic views. This elegant characterisation of secure systems has only recently been recognised as the fundamental concept underlying various definitions of non-interference, such as those surveyed in [Gra92]. Although these alternative definitions are cast in rather different notation without employing determinism, Roscoe [Ros94b] demonstrates that many are either straightforward consequence of, or closely related to, the conditions for eager, lazy, and mixed security. For example our lazy property $\mathbf{L\text{-}Sec}_H(L)$ corresponds precisely both to Graham-Cumming's own non-interference property and those of Allen [All91] and Ryan [Rya91] for systems whose overall behaviour is deterministic (as opposed to the abstractions used in formulating our properties). A significant advantage of our conditions in comparison to others is the preservation of non-interference under refinement, thus eliminating the potential compromise of security during development. A detailed discussion of this phenomenon, and an explanation of why it is desirable, may be found in [Ros94b].

The general framework envisaged for the development of secure systems falls into two parts: functional specifications of the system using state-based notations as for general applications, followed by an analysis of non-interference properties of a process-algebraic representation of the system. The main advantage of this method is in avoiding the complex treatment of establishing a correspondence between the specification and a separate generic security model. In contrast, the mapping of the specification into process algebra can in many cases be carried out by direct translation (tool support for this task is, however, at present not available). Process algebras such as CSP based on possible sequences (traces) of events provide an ideal notation for non-interference analysis since they naturally incorporate the notion of (non-)determinism, thus permitting the application of

the conditions of section 2. These conditions can then be automatically verified using a currently available proof tool.

Initial experience with the CSP model checker FDR [Ros94a] shows that a security analysis as illustrated in section 5 can be carried out within minutes. This result propounds the hope that the verification approach will scale up to systems of realistic size. The size of problem we can deal with will benefit from the proposed development [Ros94a] of FDR to incorporate *implicit* model-checking techniques such as the hierarchical compression of intermediate state-spaces. Verification speed will further increase by the exploitation of behavioural independence of processes from particular values of data communicated. This property of *data-independence* [RMacC94] has already shown promise in significant reduction of state spaces as well as the induction of properties of arbitrary data types based on finite checks.

Future work is required to formalise the mapping of state-based specifications to process descriptions. The techniques of [WM90, Woo94] still have to be extended to be applicable to specifications with complex semantics, and utilised to provide tool support for the translation into process algebra. We intend to apply these techniques and the framework outlined in this paper in a case study of a large-scale secure system. A further avenue of research is to explore potential applications of our determinism-based conditions for non-interference on systems with critical requirements other than security, such as in the areas of safety-critical systems, fault tolerance, and feature independence.

References

[All91] P.G. Allen. "A comparison of non-interference and non-deducibility using CSP", *Proc. 1991 IEEE Computer Security Workshop*, pp 43-54. IEEE Computer Society Press 1991.

[BKS83] R-J. R. Back, R. Kurki-Suonio. "Decentralization of process nets with centralized control", *Proc 2nd Annual Symposium on Principles of Distributed Computing*, Montreal, 1983.

[Col94] R. Collinson. "Proving Critical Properties of Functional Specifications", *Proc FME'94 Symposium*, Springer-Verlag LNCS, Barcelona, October 1994.

[Gas88] M. Gasser. *Building a Secure Computer System*, Van Nostrand Reinhold, 1988.

[Gra92] J. Graham-Cumming. *The Formal Development of Secure Systems*, Oxford University DPhil Thesis, 1992.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*, Prentice Hall 1985.

[Jac90] J. L. Jacob. "Specifying Security Properties", in C. A. R. Hoare (ed), *Developments in Concurrency and Communication*, ACM Press, 1990.

[Jon92] R. B. Jones. "Methods and Tools for the Verification of Critical Properties", in C. B. Jones, R. C. Shaw, T. Denvir (eds) *Proc 5th Refinement Workshop*, Springer Verlag, London, 1992.

[Ros93] A. W. Roscoe. "Unbounded Non-determinism in CSP", *Journal of Logic and Computation* **3**, 1993.

[Ros94a] A. W. Roscoe. "Model Checking CSP", in A. W. Roscoe (ed) *A Classical Mind*, Prentice Hall 1994.

[Ros94b] A. W. Roscoe. "CSP and Determinism in Security Modelling", in preparation.

[RMacC94] A. W. Roscoe, H. MacCarthy. "Verifying a replicated database: A case study in model-checking CSP", submitted for publication.

[Rya91] P. Y. A. Ryan. "A CSPformulation of non-interference", *Cipher*, pp 19-27. IEEE Computer Society Press, 1991.

[Spi92] J.M. Spivey, *The Z Notation: A Reference Manual* (2nd ed.), Prentice-Hall International, 1992.

[WM90] J. C. P. Woodcock, C. Morgan. "Refinement of State-based Concurrent Systems", *Proc VDM Symposium 1990*, LNCS 428, Springer Verlag.

[Woo94] J. C. P. Woodcock. "CSP Interpretations of Z Specifications", in preparation.

A CSP Reference

In the failures-divergences model of CSP, two processes are regarded as equal if they agree in their failures and their divergences:

$$P =_{FD} Q \Leftrightarrow \textsc{Fails}(P) = \textsc{Fails}(Q) \land \textsc{Divs}(P) = \textsc{Divs}(Q)$$

When a process Q is more deterministic than another process P we say that P is refined by Q. This relation is written $P \sqsubseteq Q$ and formally defined by

$$P \sqsubseteq Q \Leftrightarrow \textsc{Fails}(P) \supseteq \textsc{Fails}(Q) \land \textsc{Divs}(P) \supseteq \textsc{Divs}(Q)$$

The semantics of the hiding operator in the failures-divergences model is given by

$$\begin{aligned}
&\textsc{Divs}(P \setminus A) = \{ (s \setminus A)^\frown t \mid s \in \textsc{Divs}(P) \} \cup \\
&\quad \{ (s \setminus A)^\frown t \mid (\forall n \in \mathbb{N} \bullet (\exists u \in A^* \bullet \#u > n \land s^\frown u \in \textsc{Traces}(P))) \} \\
&\textsc{Fails}(P \setminus A) = \{ (u, X) \mid u \in \textsc{Divs}(P \setminus A) \} \cup \\
&\quad \{ (s \setminus A, X) \mid (s, X \cup A) \in \textsc{Fails}(P) \}
\end{aligned}$$

The semantics of $|||$ interleaving is defined as

$$\begin{aligned}
&\textsc{Divs}(P \,|||\, Q) = \{ u \mid \exists s, t \bullet u \text{ interleaves } (s, t) \land \\
&\qquad\qquad (s \in \textsc{Divs}(P) \land t \in \textsc{Traces}(Q)) \lor \\
&\qquad\qquad (s \in \textsc{Traces}(P) \land t \in \textsc{Divs}(Q)) \} \\
&\textsc{Fails}(P \,|||\, Q) = \{ (u, X) \mid u \in \textsc{Divs}(P \,|||\, Q) \} \cup \\
&\quad \{ (u, X) \mid \exists s, t \bullet u \text{ interleaves } (s, t) \land \\
&\qquad\qquad (s, X) \in \textsc{Fails}(P) \land (t, X) \in \textsc{Fails}(Q) \}
\end{aligned}$$