A security proof system for networks of communicating processes

Jean-Pierre Banâtre, Ciaran Bryce

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Security Proof System for Networks of Communicating Processes

Jean-Pierre Banâtre, Ciarán Bryce

N° 2042

septembre 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

ISSN 0249-6399

INRIA
RENNES

A Security Proof System for Networks of Communicating Processes

Jean-Pierre Banâtre, Ciarán Bryce *

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet LSP

Rapport de recherche n ° 2042 — septembre 1993 — 55 pages

Abstract: Information flow control mechanisms detect and prevent transfers of information which violate the security constraints placed on a system. In this paper we study a programming language based approach to flow control in a system of communicating processes. The language chosen to present these ideas is CSP. We give the "security semantics" of CSP and show, with the aid of two examples, how the semantics can be used to conduct security proofs of parallel programmes.

Key-words: information flow security, security variables, communicating sequential processes, axiomatic semantics, security proof system.

(Résumé : tsvp)

*email :jpbanatre, bryce@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Un système de preuve de sécurité pour Réseaux de processus communicants

Résumé : Les mécanismes de contrôle de flux d'informations ont pour rôle de détecter d'éventuels transferts d'information pouvant compromettre la sécurité attendue d'un système. Nous examinons ici le problème de contrôle de flux dans les systèmes de processus parallèles communicants. Le langage support de l'étude est CSP. Nous donnons une sémantique "sécuritaire" de CSP et montrons sur quelques exemples comment cette sémantique peut être utilisée pour construire des preuves de sécurité.

Mots-clé : flux d'information, parallélisme, sémantique axiomatique, preuves de sécurité, processus communicants.

1 Introduction

The most important aspect of computer security is the non-disclosure, or *secrecy*, of the information stored in a system. [Saltzer & Schroeder.75] define a secrecy violation as occurring when

> "An unauthorised person is able to read or take advantage of information stored in the computer"

Ensuring secrecy is equivalent to saying that certain information transmissions, or *flows*, between system objects must be prohibited. There are many examples of this: a tax programme must not leak (let flow) the private information it accesses to unauthorised processes in the system. A bank application has particularly strict secrecy requirements: information about a client's account may flow to the teller objects but not to other clients.

So what kind of mechanisms are needed to achieve information secrecy? Consider a mail application; each user process has a letter box object. One might consider two secrecy policies. Firstly, only the owning user is allowed to retrieve the contents of his letter box. Secondly, if a user A sends a message to user B marked FYEO (signifying For Your Eyes Only), then B must not be able to forward the contents of that message to some other user.

Access controls are universally used to reduce the information flows in a system [Lampson71,74, Neuman91]. To execute an operation on an object, the calling process must possess a *key* for the operation. Each process is given a set of keys, one for each of the operations in the other objects that it may legally invoke. All operating systems use this abstraction. In Unix, the key abstraction is implemented with access control lists. If a user's process has a read key for a file then the r mode bit for the file will be set for "world" or for that user's group. Capability based systems are a more direct implementation of the key abstraction [Levy84].

Going back to the mail application example, access controls can be used to enforce the secrecy requirement that only the owning user reads his own letter box, by only granting that user a key for the retrieve operation on the letter box. However, the problem arises with the second secrecy requirement. A user might receive a FYEO message and forward the contents in another message to some other user. Access controls do not understand how information flows in a system. The only way that access controls can stop a user A illegally forwarding a message to user B directly, or indirectly via other user objects, is to ensure that there is no sequence of user processes in the system starting with A and finishing with B such that all members of the sequence have a

send key for the next member's letter box. Unfortunately, this solution would prohibit A from sending **anything** to B; hence, the access control solution is insufficient.

We need some other approach for preventing undesired transmission of information between programme objects. An *information flow* mechanism must tag each variable with the set of variables from which it has received information flows, the effects of which have not been lost due to subsequent flows. The flow security of the system can then be determined from these tags. Such a mechanism requires that the behaviour of each construct of the language used to programme the system be precisely defined with respect to the information flows it generates. One can use the resulting semantics to statically analyse the programme text for flows and reject the programme if any of these flows violate the security constraints placed on the programme. This analysis can be done by hand or automatically (by a compiler for example) if enough information is available. Alternatively, the semantics can be used to specify a dynamic mechanism. This consists of extra programme instructions that a compiler inserts so that the programme information flows are logged at runtime. The approach supposes that an attacker has the programme text; he/she thus understands the expected behaviour of the programme and so may be able to deduce information by examining the value of some variable.

Information flow control mechanisms have traditionally used *security levels* [Denning75, Reitman78, Mizuno & Oldehöeft.88]. Each variable is assigned a level denoting the sensitivity of the information it contains. After an operation, the level of the variable which received the information flow must be no less than the level of the flow source variables. However, the security level approach severely restricts the range of policies that one might like to support. A flow mechanism should log the variables that have flown to each variable rather than the level of the data. [Jones & Lipton.75]'s *surveillance set* mechanism is in this spirit and has some similarities with the mechanism proposed here this paper. The differences will be discussed later.

This paper considers information flow control in systems of parallel communicating processes. We concentrate primarily on a static flow control mechanism in the form of a proof system. The goal is to be able to formally verify information flow properties from the programme text. CSP [Hoare78] is chosen to illustrate the approach since it seems a good vehicle for describing systems. The layout of the paper is as follows. A brief overview of CSP is presented in the next section. Section 3 looks at the information flow analysis of the sequential CSP language constructs and parallel composition is studied in section 4. In both sections, each command is analysed for flows

and the (axiomatic) semantics of the command regarding how it effects information flows is presented. This enables flow security proofs, that is, for any policy which the programmer may care to define, the proof system can be used to determine if the programme state at some point satisfies this policy. In particularly, after considering how a process *observes* a remote process' variables, we will use a weaker information flow consistency to capture inter-process information flows. Section 5 shows the unification of our approach and the traditional security level approach to information flow and also that flow policies may be expressed in coarser terms than inter-variable transmissions. The paper's conclusions are contained in section 6 along with a look at related work. A formal justification of the semantics is given in the appendix.

2 Communicating Sequential Processes

CSP [Hoare78] is the parallel programming language which is analysed for information flow in the following discussion. The main declarations are given in the following table.

Prog	::=	[label::Process // ..... // label::Process]	*program*
Process	::=	Decl $\prec$; Decl $\succ$; Cmd $\prec$; Cmd $\succ$	*process*
Decl	::=	**var** v | **array** v	*declarations*
Cmd	::=	Comms | Alt | Rep | **skip** | v := E | Cmd; Cmd	*commands*
Comms	::=	send | receive	*communication*
send	::=	label ! E	*send*
receive	::=	label ? v	*receive*
Alt	::=	[guard $\rightarrow$ Cmd $\prec$; $\Box$ guard $\rightarrow$ Cmd $\succ$]	*alternative*
Rep	::=	*[guard $\rightarrow$ Cmd $\prec$; $\Box$ guard $\rightarrow$ Cmd $\succ$]	*repetitive*
guard	::=	B | Comms	*guard*

where $\prec\succ$ signifies zero or more repetitions of the enclosed syntactical units, 'v' stands for a variable or a list of variables, 'E' for an integer expression and 'B' for a Boolean expression.

A CSP programme contains a fixed number of processes, each identified by a character string 'label'. This 'label' may also be an array where the entry label[i] names a process which has the same code as the other processes named in the array. Processes communicate by two-way *rendezvous*: each process names the process that it wants to communicate with; when one party in the communication executes its communication command, it blocks until the partner process is ready to execute its communication. The effect of the communication is to assign the value of the expression evaluated in the sending process to the variable in the receiving process named in the receive command.

The alternative and repetitive commands consist of one or more guard branch pairs. A guard is a Boolean expression, a communications command or a Boolean expression followed by a communication. A guard is *passable* if, for an expression, it evaluates to true, and for a communication, the process named in the command is ready to communicate. For guards consisting of both a Boolean expression and a communications command, the expression must be true and the process named in the guard must be ready to communicate for the guard to be passable.

When an alternative command is executed, a branch whose guard is passable is chosen. If more than one guard is passable, then any one of the corresponding branches can be executed. If no guards are passable then the process blocks until one of the communication guards becomes passable - unless there are no communication guards or the processes named in the guards are terminated, in which case the command fails and the process terminates.

On each iteration of the repetitive command, a branch whose guard is passable is executed. If more than one guard is passable, then like for the alternative, any one of the branches is chosen. When no guard is passable the command terminates and the process continues.

3 Information Flow in Sequential Programmes

There are two classes of information flows in programmes. An assignment command causes a **direct** flow of information from the variables appearing on the right hand side of the (:=) operator to the variable on the left hand side. This is because the information in each of the right hand side operands can influence by causing variety in the left hand side variable [Cohen77]. The information that was in the destination variable is lost.

Conditional commands introduce a new class of flows [Denning75]. The fact that a command is conditionally executed transfers information to an observer on the value of the command guard. Consider the following programme segment. We use a multiple assignment for brevity; $e()$ is some expression:

$$
\begin{array}{l}
x := e(); \\
a, b := 0, 0; \\
[\ x = 0 \rightarrow a := 1 \\
\quad \Box\ x \neq 0 \rightarrow b := 1\ ]
\end{array}
$$

If someone knows the programme text then, by inspecting either a or b after programme execution, someone can know whether x was zero or not. This is

an example of an implicit flow [Denning75] or what we will more generally refer to as an **indirect** flow.

This section examines the information flows effected by the principal sequential constructs - assignment and the alternative and repetitive commands. The notation used throughout the paper is explained beforehand.

3.1 Notation

We firstly need some way of representing the set of variables information concerning which has flown to, or influenced, a variable v. We call this set the **security variable** of v, denoted $\overline{v}$.

Indirect flows are modeled by the *indirect* variable - defined as a sequence of sets of variables:

$$indirect : (\mathbf{P}variables)^+$$

(where **P** is the "set of" operator and $^+$ stands for the set of non-zero length sequences.) The empty or nil indirect is $\prec \{\} \succ$. The operators on *indirect* are now explained.

The value of the flow of *indirect*, denoted $val(indirect)$ is the set of all variables in the indirect variable. Let $indirect(i)$ denote the i^{th} of n entries:

$$val(indirect) \triangleq \cup_{i=0}^{n-1}\ indirect(i)$$

where $\cup$ is the set union operator. Since *indirect* is just a sequence (of sets), we assume the $head()$, $tail()$ and concatenation ($\circ$) operators. A set of variables V may also be added to, as opposed to concatenated with, *indirect*. This is done with the $\uplus$ operator. The set V is set unioned with each entry in the *indirect* sequence.

$$V \uplus indirect \triangleq \circ_{i=0}^{n-1}\ (V \cup indirect(i)\ )$$

Finally, we will need an operator for combining two *indirect* variables together. The operator is $\sqcup$. Note that it is non-commutative. Its effect is to add using the $\uplus$ operator the variables in both *indirects* to the first *indirect* argument's entries:

$$indirect_i \sqcup indirect_j \triangleq (val(indirect_i) \cup val(indirect_j)) \uplus indirect_i$$

The *flow security state* of a programme is defined as firstly, the mapping from each variable to its security variable and secondly, the value of $indirect$. We stress that all the axioms and rules given in this paper are in terms of the flow security state, **not** the functional state (mapping from variables to values). When describing the behaviour of the command types with respect to the flow security state, we will use an operational notation similar to [Plotkin83]. The flow semantics are defined as a transition relation "$\rightarrow$" which maps a programme segment and state pair to another. The interpretation given $(C_1, \sigma) \rightarrow (C_2, \tau)$ is that the execution of the command sequence C_1 in state σ leads to a state τ from which the command sequence C_2 executes. Composition is specified with the following rule:

$$\frac{(S_1, \sigma) \rightarrow (\epsilon, \tau)}{(S_1; S_2, \sigma) \rightarrow (S_2, \tau)}$$

where ϵ denotes the empty command sequence.

3.2 The Assignment command

The command $y := \exp(x_1, ..., x_N)$ has the effect of setting the security variable $\overline{y}$ to

$$\{x_i \mid i = 1..N\} \cup \{\overline{x_i} \mid i = 1..N\} \cup val(indirect)$$

The term $\{x_i \mid i = 1..N\}$ captures the fact that the value of y now gives more information about what was in each x_i.

The term $\{\overline{x_i} \mid i = 1..N\}$ capture the transitivity of the information flows. For example, the programme [$b := a$; $c := b$] causes a flow from variable a and b to variable c since c contains the value of both a and b.

Constants are ignored in the flow calculus since they give no information concerning the values of variables. Their security variable is always the empty set {}. Thus the assignment a:=0 sets $\overline{a}$ to $val(indirect)$.

As mentioned, $val(indirect)$ holds the value of the information flows which the left hand side variable will indirectly receive. Indirect flows are looked at shortly in the context of the repetitive and alternative command.

A result of this approach is that a variable x may be a member of the security variable $\overline{y}$ even though y cannot be used to infer the current value of x; a subsequent assignment may have altered x. This is intentional. As long as the current value of y is functionally dependent on a (perhaps former) value of x, then x will be in $\overline{y}$. We are trying to model the fact that y has

received information form a source which may be forbidden to it, whatever the current information content of that source. Of course, when y is reset or takes an assignment not involving x, then x is no longer member of the security variable $\overline{y}$ since y's new information content is independent of (any previous) content of x.

The effect of the assignment on the programme security state is captured by the following axiom (à la [Hoare69])[1]:

$A_{:=_s}$

$$\{\ \mathrm{P}[\overline{y} \leftarrow (\{x_i \mid i = 1..N\} \cup \{\overline{x_i} \mid i = 1..N\} \cup val(indirect))]\ \}$$
$$y := exp(x_1, x_2, ..., x_N)$$
$$\{\ \mathrm{P}\ \}$$

where $P[a \leftarrow b]$ is a predicate equivalent to P except that every free occurrence of the variable a is replaced by expression b.

For array variables, one could imagine having a security variable for each element. From the point of view of a static security analysis though, it is easier to assign a single flow variable to the whole array. For an array a, index variable i and variable e, the assignment $a[i] := e$ transfers information about e and i to a since one can more easily infer the value of the two former variables from a after the assignment. (The index flows to a since its value can be known by noting the index of the element updated).

$A_{:=_s}$'

$$\{\ \mathrm{P}[\overline{a} \leftarrow \{a, e, i\} \cup \overline{a} \cup \overline{e} \cup \overline{i} \cup val(indirect)]\ \}$$
$$a[i] := e$$
$$\{\ \mathrm{P}\ \}$$

For the opposite assignment, $e := a[i]$, there is a direct flow from variables a and i to e. The variable a flows since one discovers more about it; i flows because one can know which array element was assigned from the new value of e, and hence the value of i.

$A_{:=_s}$"

$$\{\ \mathrm{P}[\overline{e} \leftarrow \{a, i\} \cup \overline{a} \cup \overline{i} \cup val(indirect)]\ \}$$
$$e := a[i]$$
$$\{\ \mathrm{P}\ \}$$

Of course, both $A_{:=_s}$' and $A_{:=_s}$" are a special case of $A_{:=_s}$: the commands $a[i] := e$ and $e := a[i]$ are equivalent to $a := exp(a, i, e)$ and $e := exp(a, i)$ respectively.

[1] All our axioms and rules have an 's' (for secure) attached to their subscript to emphasise that they are defined on the flow security state.

3.3 The Sequential command

The rule for sequential composition (;) follows. If the command S1 establishes a state satisfying predicate Q from a state P and S2 establishes R from Q, then S1 followed by S2 must establish a state satisfying R from P:

$R_{s-composition}$

$$\frac{\{\text{P}\}\ \text{S1}\ \{\text{Q}\},\ \{\text{Q}\}\ \text{S2}\ \{\text{R}\}}{\{\text{P}\}\ \text{S1;S2}\ \{\text{R}\}}$$

3.4 The Alternative command

The variables in a guard flow indirectly in the branch when it executes since execution of the branch means that the guard must be true. Moreover, a guard being true may imply that another guard is true (or false). Thus, we consider that there is an indirect flow from all guards to the branch that executes. Similarly, if a branch is not executed, then this could mean that its guard is false and therefore other guards are true (or false). Thus, there is an indirect flow from all guards to all branches which are not executed. As an example of this, consider the following programme segment.

$$\begin{array}{l} x, y := 0, 0; \\ [\ B \rightarrow x := 7; y := 9 \ \Box\ \textbf{not}\ B \rightarrow y := 2\ ]; \\ [\ x = 7 \rightarrow S_1 \ \Box\ y = 9 \rightarrow S_2 \ \Box\ x = 0 \rightarrow S_3\ ]; \end{array}$$

In the second alternative command, the execution of any branch gives information on all of the guards. That is, if S_1 executes then the condition $x = 7$ must have been true. Consequently, we know that y is 9 from the first alternative statement. This information is also discernible by observing that S_3 has not executed. Execution of S_3 implies that x is zero and that therefore y is 2 since the second branch must have executed in the preceding alternative.

The behaviour of a secure alternative command with respect to the flow security state can be described as follows. For a given flow security state σ,

$$([\ \text{i} = 1..\text{N}\ \Box C_i \rightarrow S_i;\ ],\ \sigma) \rightarrow (\textbf{update1};S_i;\textbf{update2},\sigma)$$

where

update1 $\triangleq$
$indirect := \{\ c \cup \overline{c} \mid c \in C_{bool}\ \} \circ indirect;\ \overline{l} := \overline{l} \cup \{\ c \cup \overline{c} \mid c \in C_{bool}\ \}\ \forall\ l \in lhs\_vars$
update2 $\triangleq$
$indirect := tail(indirect)$

lhs_vars is the set of variables appearing on the left hand side of the := operator in the branches of the command; C_{bool} is the set of variables appearing in the guards. The branch executed depends on the functional state.

The transition is easily explained. The indirect flows from the command guards exist only during the command body. Thus, *indirect* is updated on entry (**update1**) with the new indirect flow value which is removed on exit (**update2**). Since the variables in the branches not executed do not see the effects of *indirect*, all variables that can receive assignments in the command, *lhs_vars*, have their security variables updated with the flow value of the guard variables on entry (**update1**).

A rule describing the semantics of the alternative command is the following. We let $\overline{C_{bool}} = \{c \cup \overline{c} \mid c \in C_{bool}\}$;

$R_{s-alternative}$

$$\frac{\begin{array}{c}\mathrm{P} \Rightarrow \mathrm{R}[indirect \leftarrow \overline{C_{bool}} \circ indirect, \overline{l} \leftarrow \overline{l} \cup \overline{C_{bool}}\ \forall\ l \in lhs\_vars], \\ \forall i = 1..N\ \{\ \mathrm{R}\ \}\ \mathrm{S}_i\ \{\ \mathrm{T}\ \}, \\ \mathrm{T} \Rightarrow \mathrm{Q}[indirect \leftarrow tail(indirect)]\end{array}}{\{\ \mathrm{P}\ \}\ [\ C_1 \rightarrow S_1\ \Box\ C_2 \rightarrow S_2\ \Box\ .....\ \Box\ C_N \rightarrow S_N\ ]\ \{\ \mathrm{Q}\ \}}$$

The rule is geared towards flow security proofs. Hence, one wants a predicate T that is established no matter which of the command branches S_i is executed. It should be re-emphasised that the rule says nothing about the values of the command guards since it deals only with the flow security state.

Finally, note that the case of the process terminating due to all guards failing is not dealt with. Such an event does not cause any flows between programme variables. However, information can be leaked to the environment in a runtime version of the flow mechanism. We return to this point in section 5.2.

3.5 The Repetitive command

Repetitive commands also cause indirect information flows from the variables in the guards to all variables which could possibly receive flows in the loop body since we can know the value of the guards by examining the variables in the loop - even if the loop does not execute.

$x, r := e_1(), e_2();$ /* *expressions return non-negative values* */
$z, y, t := 0, 0, 0;$
$*[\ x \neq y \rightarrow y := y + 1$
$\quad \Box\ r \neq t \rightarrow t := t + 1\ ];$
$z := 1;$

In this programme segment, the values of y and t will equal x and r respectively on loop termination, even if none of the branches execute.

Moreover, as [Reitman78] pointed out, since all variables receiving direct flows after the loop do so on condition that the loop terminates, the variables of the loop guards flow indirectly to these variables. This is because it is known after the loop termination that all guards are false. In the example above, by observing that z is 1 one knows that x equals y and r equals t. Consider thus the behaviour of the repetitive command with respect to the flow security state.

$$(*[\ \mathrm{i} = 1..\mathrm{N}\ \Box C_i \rightarrow S_i;\ ],\ \sigma) \rightarrow (\textbf{update 1};S_i;\textbf{update2};*[\ \mathrm{i} = 1..\mathrm{N}\ \Box C_i \rightarrow S_i;\ ],\sigma) \text{ or } (\textbf{update3},\ \sigma)$$

where

update3 $\triangleq$
$indirect := \{\ c \cup \overline{c} \mid c \in C_{bool}\ \} \uplus indirect;\ \overline{l} := \overline{l} \cup \{\ c \cup \overline{c} \mid c \in C_{bool}\ \}$

On each iteration of the loop, the flow value of the loop guard is pushed onto the *indirect* stack (**update1**) and popped at the end (**update2**). When the loop finally terminates, the indirect flow from the loop conditions to all variables that could have received a flow in the loop (*lhs_vars*, to cater for the case when no branch executes) and all variables which subsequently receive a flow is recorded (**update3**). Note how the $\uplus$ operation is used instead of the $\circ$ to capture the permanence of the change in *indirect*; moreover, the number of entries in *indirect* is the same on entry and exit since any arbitrary nesting scheme of alternative and repetitive commands must be supported.

A rule for how the secure repetitive command influences the flow security state is:

$R_{s-repetitive}$

$$\frac{\begin{array}{c}\mathrm{P} \Rightarrow \mathrm{R}[indirect \leftarrow \overline{C_{bool}} \circ indirect,\ \overline{l} \leftarrow \overline{l} \cup \overline{C_{bool}}\ \forall\ l \in lhs\_vars],\\ \mathrm{R} \Rightarrow \mathrm{P}[indirect \leftarrow tail(indirect)],\\ \forall i = 1..N\ \{\ \mathrm{R}\ \}\ \mathrm{S}_i\ \{\ \mathrm{R}\ \},\\ \mathrm{P} \Rightarrow \mathrm{Q}[indirect \leftarrow \overline{C_{bool}} \uplus indirect,\ \overline{l} \leftarrow \overline{l} \cup \overline{C_{bool}}\ \forall\ l \in lhs\_vars]\end{array}}{\{\ \mathrm{P}\ \} *[\ C_1 \rightarrow S_1\ \Box\ C_2 \rightarrow S_2\ \Box\ .....\ \Box\ C_N \rightarrow S_N\ ]\ \{\ \mathrm{Q}\ \}}$$

We justify this rule using the transition given above. Since the command body may be executed any number of times, we need an invariant on the flow security state. P serves as this invariant. Moreover, the modifications that occur to the *lhs_vars* and *indirect* variables at the start and end of each branch allow an *inner* invariant R to be established[2]. After termination, the final modifications to the flow variables establish a state satisfying Q.

[2]In fact, this does not have to be an invariant just as long as P is re-established at the end of the iteration; nevertheless, it is convenient to think of R as being invariant.

A proof of the soundness and completeness of the proof system is given in the appendix along with a proof that the semantics capture all the information flows.

3.6 A Flow Security Proof Example

The example we will consider is a library decryption programme. The programme has three inputs and two outputs as shown in the diagram. The input consists of a string of encrypted text, or *cipher-text*, a key for decryption and a unit rate which the user is charged for each character decrypted. The outputs are the decrypted text, or *clear-text*, and the charge for the decryption. We assume that the clear-text is output to the user and that the charge is output to the library owner. To be usable, the user must trust the programme not to secretly leak the clear-text to the library owner via the charges output. Such a leakage is termed a *covert channel* in [Lampson73]. We will therefore flow secure prove the decryption programme, the code of which is given below.

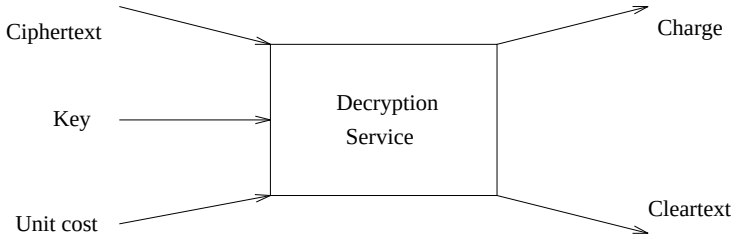

Figure 1: Decryption Schema

---

var: i, *charge*, *key*, *unit*;
array: *clear_text*, *cipher_text*;
cipher_text := $\prec$ message to be decrypted $\succ$;
unit := $\prec$ unit rate constant $\succ$;
charge := *unit*;
$i := 0$;
$*[$ *cipher_text*$[i] \neq$ null_constant $\rightarrow$
[encrypted(*cipher_text*$[i]) \rightarrow$ *clear_text*$[i]$:= D(*cipher_text*$[i]$, *key*);
charge := *charge* + 2**unit*;
$\Box$ **not** encrypted(*cipher_text*$[i]) \rightarrow$ *clear_text*$[i]$:= *cipher_text*$[i]$;
charge := *charge* + *unit*;
];
$i := i + 1$;
]

The cipher-text is an array of characters. A character is decrypted by applying it to an expression D with the *key* parameter. To save computing resources, some characters may not have been encrypted. The user pays twice the price for every encrypted character that goes through the decryption programme. The Boolean expression *encrypted()* determines if the character passed is encrypted or not. □

A comment on the notation: In the proof that follows, rather than re-writing large formulae, we will use the following notation for brevity:

$$\text{P } \textbf{but } \alpha = \{\beta\}$$

is an assertion which is similar to P except that the flow value α is now $\{\beta\}$. Similarly,

$$\text{P } \textbf{but } \alpha \textbf{ removed}$$

is an assertion which is similar to P except that there is no sub-predicate for α. For completeness we give the consequence rule:

$R_{consequence}$

$$\frac{\text{P}\Rightarrow\text{P'},\ \{\text{P'}\}\ \text{S}\ \{\text{Q'}\},\ \text{Q'}\Rightarrow\text{Q}}{\{\text{P}\}\ \text{S}\ \{\text{Q}\}}$$

The flow security proof of the programme is as follows. We let the security flow values of the input variables *unit*, *cipher_text* and *key* be denoted by $\vec{unit}$, $\vec{cipher}$ and $\vec{key}$ respectively. (We will write "cipher" for "cipher_text" and "clear" for "clear_text" for reasons of brevity).

We formalise the post condition as follows:

$$post\_condition \equiv \{\ clear \notin \overline{charge},\ \vec{key} \notin \overline{charge}\ \}$$

that is, the charge output may not receive a flow of information from the clear-text variable or from the key input.

With the precondition
PRE ≡

$$\{\ \overline{clear} = \{\},\ \overline{charge} = \{\},\ \overline{cipher} = \vec{cipher},\ \overline{key} = \vec{key},\ \overline{unit} = \vec{unit},\ \bar{i} = \{\},\ indirect = \prec\ \{\}\ \succ\ \}$$

We are going to establish the following condition:
POST ≡

$$\{\ \overline{clear} \subseteq \{clear, cipher, \vec{cipher}, i, key, \vec{key}\},\ \overline{charge} \subseteq \{cipher, \vec{cipher}, i, charge, unit, \vec{unit}\},\ \overline{cipher} = \vec{cipher},\ \overline{key} = \vec{key},\ \overline{unit} = \vec{unit},\ \bar{i} \subseteq \{cipher, \vec{cipher}, i\},\ indirect = \prec\ \{cipher, \vec{cipher}, i\}\ \succ\ \}$$

Regarding POST, the clear-text will depend on the cipher-text and the key. The charges output is permitted to receive flows from the cipher-text and from the unit charge. It is easy to show that POST $\Rightarrow$ *post_condition.*

The plan of the flow security proof is as follows. $\mathcal{P}$ is the repetitive command invariant and pre-condition. PRE is the programme pre-condition and POST its post-condition:
Plan :

i) initialisation statements establish $\{\mathcal{P}\}$ from {PRE}
ii) {POST} is established when the loop terminates
iii) the predicate $\{\mathcal{P}\}$ is invariant over all iterations

The following is chosen as the loop invariant $\mathcal{P}$:

$$\text{POST } \textbf{but } \mathit{indirect} = \prec \{\} \succ$$

i) Proof of: {PRE} *charge := unit; i := 0* $\{\mathcal{P}\}$
$i := 0$;
let P' be $\mathcal{P}$ but with updates associated with assignment
P' = $\mathcal{P}[\overline{i} \leftarrow \mathit{val}(\mathit{indirect})]$
and since *indirect* is empty in $\mathcal{P}$,
P' = $\mathcal{P}[\overline{i} \leftarrow \{\}]$.
Since $\{\} \subseteq \{\mathit{cipher}, \vec{\mathit{cipher}}, i\}$, the assertion for $\overline{i}$ is removed, so we get P':

$$\mathcal{P} \textbf{ but } \overline{i} \textbf{ removed}$$

charge := *unit*;
Let P" be P' but with updates associated with assignment
$\equiv$ P'$[\overline{\mathit{charge}} \leftarrow \{\mathit{unit}\} \cup \overline{\mathit{unit}} \cup \mathit{val}(\mathit{indirect})]$
$\equiv$ P'$[\overline{\mathit{charge}} \leftarrow \{\mathit{unit}\} \cup \vec{\mathit{unit}} \cup \{\}]$
and so the sub-predicate for *charge* becomes:
$\{\mathit{unit}, \vec{\mathit{unit}}\} \subseteq \{\mathit{cipher}, \vec{\mathit{cipher}}, i, \mathit{charge}, \mathit{unit}, \vec{\mathit{unit}}\}$

$$\equiv \text{P' } \textbf{but } \overline{\mathit{charge}} \textbf{ removed}$$

So it has been shown that {P"} *charge := unit; i := 0* $\{\mathcal{P}\}$
Since it can easily be shown that PRE $\Rightarrow$ P"
$\Rightarrow$ {PRE} *charge := unit; i := 0* $\{\mathcal{P}\}$ by $R_{consequence}$
qed.

ii) We want to show that {POST} is established when the loop terminates. Since no commands follow the loop, POST is equivalent to the repetitive command post-condition Q.
We let $\overline{C_{bool}}$ represent $\{c \cup \overline{c} \mid c \in C_{bool}\}$
We must show $\mathcal{P} \Rightarrow$ Q$[\mathit{indirect} \leftarrow \overline{C_{bool}} \uplus \mathit{indirect}, \overline{\mathit{lhs_vars}} \leftarrow \overline{\mathit{lhs_vars}} \cup \overline{C_{bool}}]$
(from $R_{s-repetitive}$
let Q' = Q$[\mathit{indirect} \leftarrow \overline{C_{bool}} \uplus \mathit{indirect}, \overline{\mathit{lhs_vars}} \leftarrow \overline{\mathit{lhs_vars}} \cup \overline{C_{bool}}]$
$\equiv$ Q$[\mathit{indirect} \leftarrow \overline{\mathit{cipher}} \cup \overline{i} \cup \{\mathit{cipher}, i\} \uplus \mathit{indirect}, \overline{\mathit{clear}} \leftarrow \overline{\mathit{clear}} \cup \overline{\mathit{cipher}} \cup \overline{i} \cup \{\mathit{cipher}, i\}, \overline{i} \leftarrow \overline{i} \cup \overline{\mathit{cipher}} \cup \overline{i} \cup \{\mathit{cipher}, i\}, \overline{\mathit{charge}} \leftarrow \overline{\mathit{charge}} \cup \overline{\mathit{cipher}} \cup \overline{i} \cup \{\mathit{cipher}, i\}]$

$$\begin{array}{l}\text{Q' } = \{\ \overline{clear} \cup \overline{cipher} \cup \overline{i} \cup \{cipher, i\} \subseteq \{clear, cipher, \vec{cipher}, i, key, \vec{key}\},\\ \overline{charge} \cup \overline{cipher} \cup \overline{i} \cup \{cipher, i\} \subseteq \{cipher, \vec{cipher}, i, charge, unit, \vec{unit}\},\\ \overline{cipher} = \vec{cipher},\ \overline{key} = \vec{key},\ \overline{unit} = \vec{unit},\\ \overline{i} \cup \overline{cipher} \cup \overline{i} \cup \{cipher, i\} \subseteq \{cipher, \vec{cipher}, i\},\\ \overline{cipher} \cup \overline{i} \cup \{cipher, i\} \uplus indirect = \prec \{cipher, \vec{cipher}, i\} \succ\ \}\end{array}$$

$$\equiv \text{Q } \mathbf{but}\ indirect \subseteq \prec \{cipher, \vec{cipher}, i\} \succ\ \}$$

$\mathcal{P} \Rightarrow$ Q', since the assertions only differ in *indirect* and $indirect = \prec \{\} \succ \Rightarrow indirect \subseteq \prec \{cipher, \vec{cipher}, i\} \succ$. So, by $R_{consequence}$,
$\mathcal{P} \Rightarrow \text{Q}[indirect \leftarrow \overline{C_{bool}} \uplus indirect, \overline{lhs\_vars} \leftarrow \overline{lhs\_vars} \cup \overline{C_{bool}}]$
qed.

(iii) Proof that $\mathcal{P}$ is invariant.
Using $R_{s-repetitive}$, we can let $\mathcal{R}$ be:

$$\begin{array}{l}\{\ \overline{clear} \subseteq \{clear, cipher, \vec{cipher}, i, key, \vec{key}\},\ \overline{charge} \subseteq \{cipher, \vec{cipher}, i, charge, unit, \vec{unit}\},\\ \overline{cipher} = \vec{cipher},\ \overline{key} = \vec{key},\ \overline{unit} = \vec{unit},\\ \overline{i} \subseteq \{cipher, \vec{cipher}, i\},\ indirect = \prec \{cipher, \vec{cipher}, i\}\{\} \succ\ \}\end{array}$$

iii.i) $i := i + 1$;
Let R' be the assertion which is held prior to the assignment. From $A_{:=_s}$
$\text{R'} = \mathcal{R}[\overline{i} \leftarrow \overline{i} \cup \{i\} \cup val(indirect)] = \mathcal{R}$ **but** $\overline{i} \cup \{i\} \cup \{cipher, \vec{cipher}, i\} \subseteq \{cipher, \vec{cipher}, i\}$
$\equiv \mathcal{R}$.

iii.ii) We now prove the alternative command. We note that in our case the command pre-condition and post-condition are both $\mathcal{R}$.
We must look for a T with these properties from $R_{s-alternative}$:
(a) $\mathcal{R} \Rightarrow \text{T}[indirect \leftarrow \overline{C_{bool}} \circ indirect, \overline{lhs\_vars} \leftarrow \overline{lhs\_vars} \cup \overline{C_{bool}}]$,
(b) $\text{T} \Rightarrow \mathcal{R}[indirect \leftarrow tail(indirect)]$
T is chosen as:

$$\mathcal{R}\ \mathbf{but}\ indirect = \prec \{cipher, \vec{cipher}, i\}\{cipher, \vec{cipher}, i\}\{\} \succ$$

To show (a)
let $\text{T'} = \text{T}[indirect \leftarrow \overline{C_{bool}} \circ indirect, \overline{lhs\_vars} \leftarrow \overline{lhs\_vars} \cup \overline{C_{bool}}]$
$\equiv \text{T}[indirect \leftarrow \{cipher, i\} \cup \overline{cipher} \cup \overline{i} \circ indirect, \overline{charge} \leftarrow \overline{charge} \cup \{cipher, i\} \cup \overline{cipher} \cup \overline{i}, \overline{clear} \leftarrow \overline{clear} \cup \{cipher, i\} \cup \overline{cipher} \cup \overline{i}]$
This is similar to the proof conducted at the end of (ii); the predicate can be shown to be equivalent to $\mathcal{R}$.
Since $\mathcal{R} \equiv$ T' so property (a) is satisfied. It is trivial to show property (b).

We now show that both branches of the selection command preserve T.
The proof of **branch 1** is as follows:
charge := *charge* + 2**unit*;

let T' be T but with updates due to assignment; it is the predicate which must be true before the assignment.
T' = T$[\overline{charge} \leftarrow \{charge, unit\} \cup \overline{charge} \cup \overline{unit} \cup val(indirect)]$
$\equiv$ { T **but** $\overline{charge}$ **removed**,
$\overline{charge} \cup \{charge, unit, \vec{unit}, cipher, \vec{cipher}, i\} \subseteq \{charge, unit, \vec{unit}, cipher, \vec{cipher}, i\}$
}
$\equiv$ T

clear_text[i] := D(*cipher_text[i]*, *key*);
let T' be T but with updates due to assignment
T' = T$[\overline{clear} \leftarrow \{clear, cipher, i, key\} \cup \overline{cipher} \cup \overline{clear} \cup \overline{i} \cup \overline{key} \cup val(indirect)]$
$\equiv$ { T **but** $\overline{clear}$ **removed**,
$\overline{clear} \cup \{clear, cipher, \vec{cipher}, i, key, \vec{key}\} \subseteq \{clear, cipher, \vec{cipher}, i, key, \vec{key}\}$
} also equal to T
So { T } branch 1 { T }

The proof of **branch 2** is almost the same proof as for branch 1 so we omit it here.

So, both branches of the alternative command preserve T which means that $\mathcal{R}$ is indeed the pre and post condition of the alternative. Moreover, we have shown that $\mathcal{R}$ is also the "inner invariant" of the repetitive command which formally derives from $\mathcal{P}$ using $R_{s-repetitive}$. Thus $\mathcal{P}$ is invariant on the repetitive command and so part (iii) of the proof is complete. The programme is thus proved flow secure with respect to our requirements. □

It should be noted that the cipher-text will depend on the key and the original clear-text. However, the programme is still certified since this information flow dependency does not occur in the decryption programme itself.

4 Parallelism and Information Flow

This section examines information flow in a parallel framework, using the complete version of CSP [Hoare78]. The problems relating to inter-process indirect flows are first looked at; then the flows effected by each of the command types is described, accompanied by the semantics. A flow security proof of a CSP programme using the complete proof system is outlined in the final subsection. The proof system is modeled on that of [Apt et al.80] for proving the correctness of a CSP programme.

4.1 Interprocess Information Flows

With processes exchanging data, we need to consider how inter-process information flows are defined. Direct flows occur during communication. There

is a direct flow from the variables named in the expression of the send command to the variable named in the receive command.

Regarding indirect flows, each process has its own *indirect* variable. It was mentioned in the last section that there is an indirect flow from conditional command guard variables to the variables which can receive flows in the branches. In a similar way, when a process does a rendezvous, there are subsequent updates and communications with other processes. The fact that one of these communications takes place, and that the updates which follow are made, gives information to each process about the condition that was met in the process that communicated with it. This information is contained in the process' *indirect* variable which must thus be exchanged during rendezvous. If the communication is not made, then the fact that no updates occur in a remote process can be indicative of the condition for the rendezvous not being met in the first process. Thus, there is still an indirect flow in the absence of a rendezvous.

As an example of how indirect flows can occur in the absence of a rendezvous, consider the following programme segment. Suppose that y of process $P1$ is either 1 or 0. Whatever, the value of y, at the end of process $P1$, x will equal y. The reason for this is that, if $y = 0$ in process P1, then P1 passes the value 1 to b of process P2 which then passes 0 back to x. Conversely, if y is 1 in P1, then P1 signals 0 to process P3 which signals 1 to P2's b which in turn passes this value back to x. To cope with this, if $P1$ misses its communication with $P2$ in the alternative command, the flow value containing y must normally be transferred to $P2$'s b so that its subsequent rendezvous with $P1$ (when it sends a) will permit y to be registered in the flow variable of x.

```
[
        P1::                          P2::                          P3::
        [var x,y;                     [var a,b;                     [var s;
        y := e();                     [ P1 ? b → b:=b-1 □           P1 ? s;
        [ y=0 → P2 ! 1 □                P3 ? b → skip ]             P2 ! 1
          y≠0 → skip ]                a := b;                       ]
        P3 ! 0;                       P1 ! a
        P2 ? x                        ] //
        ] //
]
```

[Mizuno & Oldehöeft.88] recognised the problem of interprocess indirect flows in the context of an object-based system. In their proposal, each time a communication with another object is skipped in a conditional command, a dummy message called a *probe* is sent to all objects which could have been

transitively communicated with, had the conditional communication been made by the object. The probe carries the information flow value (in their case the security classification) of the variables in the condition of the command containing the communication. On arriving at the destination objects, the classification is added to the current classification of the variables which could have received a direct flow if the communication had gone ahead.

We believe the probe message approach to be pessimistic for flows and to have an unacceptably high cost. For a dynamic flow mechanism, sending a message each time a communication is skipped would flood the system. Another problem with it is that we found it difficult to define the semantics in CSP for the probe message exchange. This was due to the rendezvous communication: a receiver stated explicitly when it wanted a message. Thus the probe would have to be consumed at a precise point in time. Moreover, if the process receiving a flow also missed its communication, then it is not evident how and when we express the updating with the probe value. To counter the performance difficulties, [Mizuno & Oldehöeft.88] did produce a link time optimisation though this is only for cases where all the processes (objects) are known and named at compile/link time and where the security levels are statically bound to the variables. Such a solution does not suit us. Though in CSP the first criterion is met, we would like our flow semantics to be as easily adaptable to other process/programming models where the criteria may not hold. Secondly, it is not evident if a mechanism based on security variables is as easily adaptable to the link time analysis as one based on statically bound security levels.

A better solution comes from considering the causal relations that exist among processes and the way that processes view the system. A process cannot observe all the events in a system as they happen; it must be informed of these events by other processes via communication. This has important implications for the treatment of interprocess indirect flows. Consider the piece of code:

$$[\ x = 0 \rightarrow a := 1 \ \Box \ x \neq 0 \rightarrow b := 1\ ]$$

where a and b are both zero beforehand. After execution, b will give information about x. Consider what happens if the assignment $b := 1$ is in a remote process: If $x \neq 0$ then b is updated and the indirect flow from x occurs. However, if one observes b and sees that it is zero then one cannot infer that the condition failed in the first process: the alternative command may not have executed yet or P2 may have communicated with some other process. The only way that the value of the condition in P1 can be inferred from any value in P2 is if a set of communications from P1 to P2 takes place, after

```
[
  P1::                              P2::
  [                                 [
  ⋮                                 ⋮
  [ x=0 → a:=1 □                      [ ... □ P1 ? b → skip ]
    x≠0 → P2 ! 1 ]                  ⋮
  ⋮                                 ]
  ] //
]
```

the alternative has executed in P1. This also covers, for example, an observer process looking at some variable in P1 and seeing that the alternative has completed and then communicating with P2 to see whether b was updated, with the intention of deducing x.

The solution we propose is to transfer the flow value of a condition on which a communication executes, only if the communication takes place. If the communication is skipped, then the flow value of the condition is recorded in a special variable *rendezvous* (one per process). On each communication, *rendezvous* in transferred in both directions as part of the indirect flow. In our mechanism, *rendezvous* is incorporated into the *indirect* variable. This approach works because interprocess indirect flows can only signal any useful information if a process from which the flow originates, (transitively) communicates with the process with which it should have communicated, later on. With the *rendezvous* mechanism, we are guaranteed that the flow value of the condition in question will be transferred when this subsequent communication occurs. This approach constitutes what we call *weak information flow consistency.*

The weak consistency approach implies that the conditional commands do not always undo the *indirect* after each branch. Nevertheless we believe that weak information flow consistency can be less pessimistic overall in registering information flows than the probe mechanism.

4.2 CSP Flow Security Inference Rules

Using this weaker consistency approach, a security flow proof system for CSP is now presented. Proofs are conducted in two phases. In the first phase, each process is proved individually with assumptions being made on the

flow values exchanged during communication. The second stage of the proof demonstrates that the proof of the processes are consistent with one another.

4.2.1 The Send and Receive commands

The effect of the communication command is to assign the expression in the send command to the variable in the receive command. This is a direct flow. In addition, both *indirects* are updated to include each other's value using the $\sqcup$ operator since execution of both processes is now dependent on the rendezvous having taken place.

$$(P_2!x \;//\; P_1?y, \sigma) \rightarrow (\in, \sigma')$$

where σ resembles σ' except that $\overline{y}$ in σ' equals $(\overline{x} \cup x \cup val(indirect_1) \cup val(indirect_2))$, $indirect_1$ in σ' equals $(indirect_1 \sqcup indirect_2)$ of σ and $indirect_2$ is $(indirect_2 \sqcup indirect_1)$.

In accordance with our two staged approach to proving parallel programmes secure, for both the send and receive command axioms, any post-condition is allowed to be established:

A_{s-send}

$$\{\ \mathrm{P}\ \} \prec Process \succ\ ! \prec Expression \succ \{\ \mathrm{Q}\ \}$$

$A_{s-receive}$

$$\{\ \mathrm{P}\ \} \prec Process \succ\ ? \prec Variable \succ \{\ \mathrm{Q}\ \}$$

The suitability of the post-condition chosen is verified in the second stage of the proof with the help of the communication axiom:

$A_{s-communication}$

$$\frac{\{\ z_1 = indirect_1,\ z_2 = indirect_2\ \}\quad P_2!x\ //\ P_1?y}{\{\ \overline{y} = \{x\} \cup \overline{x} \cup val(indirect_1) \cup val(indirect_2),\ indirect_1 = z_1 \sqcup z_2,\ indirect_2 = z_2 \sqcup z_1\ \}}$$

There is a special case of the $A_{s-communication}$ axiom to cater for a process being named with the help of an expression - when the process' name is an array entry. For example, $P[i]!X//Q?y$ permits the receiving process to discern i since continuation of its execution depends on it. Thus, during the rendezvous, the process indices are exchanged as part of the *indirects*. More specifically, for the rendezvous,

$$P_{R[i]}!x\ //\ P_{S[j]}?y$$

we have as part of the $A_{s-communication}$ postcondition for the *indirects*:

$indirect_{R[i]} = (\{i\} \cup \overline{i}) \uplus (z_{R[i]} \sqcup z_{S[j]})$, $indirect_{S[j]} = (\{j\} \cup \overline{j}) \uplus (z_{S[j]} \sqcup z_{R[i]})$

In this case the indices are variables, but it generalises for expressions.

4.2.2 The Alternative command

The complete alternative command of CSP is more complex than that introduced in section 3. A guard may be a Boolean expression, a communications command or a Boolean expression followed by a communications command. However, in the presentation which follows, we assume the latter form $b; \alpha \rightarrow S$ to be rewritten as $b \rightarrow \alpha; S$ where b is a non-constant Boolean expression (containing variables as opposed to the constant **true** say), α a communications and S a command sequence.

As mentioned previously, when a branch with a Boolean guard executes, there is an indirect information flow from all Boolean guards to the branch variables. In contrast, a communications guard creates no such flow - one cannot know the value of some Boolean guard from the fact that a communications guard was passable and its branch executed. The indirect flows in each branch effected by the complete alternative command are now summarised:

When a branch with a Boolean guard executes, its guard variables flow indirectly in the branch since the condition must be true; the other Boolean guards also flow since the branch guard may imply that they are true or false.

All other branches receive a flow from the Boolean guards, even branches with communication guards, since the branch not executing may mean that a Boolean guard somewhere is true.

When a branch with a communications guard executes, there are no indirect flows in the command since nothing can be inferred about the values of the Boolean guards. Execution of the branch is not conditioned on the value of any of the command's Boolean guards.

The transitions for the alternative command is the following.

$$([\text{ i = 1..N } \Box C_i \rightarrow S_i;], \sigma) \rightarrow^{bool(C_i)} (\textbf{update1}; S_i; \textbf{update2}', \sigma)$$

where **update1** is the same as for the sequential version, $bool(C_i)$ is true if the guard is a Boolean expression, otherwise $comms(C_i)$. *missed_comms* is true if one of the other branches of the alternative command contains a communication with a process other than one communicated with in the executing branch. and for a branch with a communications guard.

$$([\text{ i = 1..N } \Box C_i \rightarrow S_i;], \sigma) \rightarrow^{comms(C_i)} (S_i, \sigma')$$

if there exists a matching communication command α, such that $(\alpha // C_i, \sigma) \rightarrow (\epsilon, \sigma')$.

update2' $\hat{=}$ **if** *missed_comms*
then $indirect := head(indirect) \uplus tail(indirect)$
else $indirect := tail(indirect)$
endif

The semantics is explained as follows. **update1** captures the indirect flows that occur in the branches at the start of a Boolean guarded branch - all guards' variables flow to all branches. After the branch has executed, one must undo the changes to the *indirect* stack for the branch if it has a Boolean guard (**update2'**, else part) except if a communications with some other process, not communicated with in the current branch, has been skipped. This captures the flow arising due to the weak consistency protocol (**update2'**, then part).

We derive the following rule:

$R_{s-alternative}$

$$\frac{\begin{array}{c}\textbf{if } \text{bool}(C_i) \textbf{ then } (\text{P} \Rightarrow \text{U[replace1]}, \{\text{U}\}\ S_i\ \{\text{V}\}, \text{V} \Rightarrow \text{Q[replace2]}) \\ \textbf{if } \text{comms}(C_i) \textbf{ then } (\{\ \text{P}\ \}\ C_i\ \{\ \text{T}_i\ \}, \{\ \text{T}_i\ \}\ S_i\ \{\ \text{Q}\ \})\end{array}}{\{\ \text{P}\ \}\ [\ C_1 \rightarrow S_1 \ \Box\ C_2 \rightarrow S_2 \ \Box\ ..... \ \Box\ C_N \rightarrow S_N\ ]\ \{\ \text{Q}\ \}}$$

for some predicate $\mathcal{P}$,

$\mathcal{P}$[replace1] $\hat{=}$
$\mathcal{P}[indirect \leftarrow \overline{C_{bool}} \circ indirect, \overline{l} \leftarrow \overline{l} \cup \overline{C_{bool}}, \forall\ l \in lhs\_vars]$
$\mathcal{P}$[replace2] $\hat{=}$
$missed\_comms : \mathcal{P}[indirect \leftarrow head(indirect) \cup tail(indirect)]$
not $missed\_comms : \mathcal{P}[indirect \leftarrow tail(indirect)]$

This rule is justified as follows. Consider the case of a branch with a communications guard executing. The guard can establish any predicate $\{\text{T}_i\}$. Because we are dealing with an alternative command, the branch must establish the post-condition {Q} directly since there is no change to *indirect* to be handled at the end. For branches with Boolean guards, the predicate {U} captures the changes to the security variables on entry to the branch. The branch body establishes {V} which must establish {Q} when *indirect* is handled at the end.

Note again that no functional predicates have been used, that is, predicates on the mapping from variables to values. Their use would greatly benefit the security analysis since it could allow the elimination of several guards from consideration. For example, in the following programme segment,

$$\{\ x \geq 0\ \} \text{ – predicate on functional state}$$
$$[\ x < 0 \rightarrow S_1 \ \Box\ x = 0 \rightarrow S_2 \ \Box\ x > 0 \rightarrow S_3\ ]$$

one can eliminate the first guard from analysis because we know it can never execute and produce flows. Another area where extra functional information is useful is in determining indirect flows. Some guards being true (or false) have no implication regarding other guards. There is no indirect flow from the latter guards when one of the former is true and its branch is executed. However, it is not clear how easy it would be when functionally analysing a programme to know which guards are dependent; the predicates may not be always be able to give us that information. In any case, runtime implementations of the flow mechanism would lack this information.

4.2.3 The Repetitive command

The template of the command is the following. The notation used is the same as in the alternative semantics.

$$(*[\ \text{i = 1..N}\ \Box C_i \rightarrow S_i;\ ],\ \sigma) \rightarrow^{bool(C_i)} (\textbf{update1};S_i;\textbf{update2'};*[\ \text{i = 1..N}\ \Box C_i \rightarrow S_i;\ ],\sigma) \text{ or } (\textbf{update3},\ \sigma)$$

and for a communications guard ...

$$(*[\ \text{i = 1..N}\ \Box C_i \rightarrow S_i;\ ],\ \sigma) \rightarrow^{comms(C_i)} (S_i;*[\ \text{i = 1..N}\ \Box C_i \rightarrow S_i;\ ],\sigma') \text{ or } (\textbf{update3},\ \sigma)$$

where the **updates** are as before; and σ' is a valid state reached by a communication. The semantics can be explained as follows. If the branch that executes has a Boolean guard, then there is an indirect flow from all of the Boolean guards to all branches (**update1**). The reasoning is the same as for the alternative command. After the branch is executed, the *indirect* stack is popped (**update2'**, else part) except in the case where a communication is skipped in another branch, where the indirect is re-updated using the $\uplus$ operator to account for the rendezvous flow (**update2'**, then part). Finally, after termination of the command, *indirect* is updated along with the *lhs_vars* variables, as was described in section 3, to capture the termination condition. The rendezvous flow is implicitly taken care of (**update3**) for the case where no branch is taken.

And a rule describing the semantics, derived using the same reasoning as $R_{s-alternative}$:

$R_{s-repetitive}$

$$\frac{\begin{array}{c}\textbf{if } bool(C_i) \textbf{ then } (\text{P} \Rightarrow \text{U[replace1]},\ \{\text{U}\}\ S_i\ \{\text{V}\},\ \text{V} \Rightarrow \text{P[replace2]}),\\ \textbf{if } comms(C_i) \textbf{ then } (\{\text{P}\}\ \text{C}_i\ \{\text{T}_i\},\ \{\text{T}_i\}\ S_i\ \{\text{P}\}),\\ \text{P} \Rightarrow \text{Q[replace3]}\end{array}}{\{\ \text{P}\ \}\ *[\ C_1 \rightarrow S_1\ \Box\ C_2 \rightarrow S_2\ \Box\ .....\ \Box\ C_N \rightarrow S_N\ ]\ \{\ \text{Q}\ \}}$$

where

$\mathcal{P}[\text{replace3}] \triangleq \mathcal{P}[indirect \leftarrow \overline{C_{bool}} \cup indirect, \overline{l} \leftarrow \overline{l} \cup \overline{C_{bool}}, \forall\, l \in lhs\_vars]$

4.2.4 Co-operation

The second phase of a CSP programme flow security proof demonstrates that proofs of processes P_1, P_2,....., P_N co-operate. The rule assumed is the following:

$R_{co-operation}$

$$\frac{\{pre_i\}\ P_i\ \{post_i\}\ \text{i = 1,...,N co-operate}}{\{\wedge pre_i \mid i = 1..N\}\ P_1\ //\ P_2\ //\ ....\ //\ P_N\ \{\wedge post_i \mid i = 1..N\}}$$

Processes co-operate if the following two conditions hold:

1. The predicates used in the proof of process P_i contain no free variables modifiable in any other process P_j.

2. For all semantically paired communication commands $\{p_i\}P_j! \prec expression \succ \{q_i\}$ and $\{p_j\}P_i? \prec variable \succ \{q_j\}$,

$$\{p_i \wedge p_j\}P_j! \prec expression \succ //P_i? \prec variable \succ \{q_i \wedge q_j\}.$$

Note how condition (2) says *semantically* matching communication commands, that is, commands which name each other's process and which can possibly communicate during programme execution. It may seem strange that semantically paired commands are stated in the rule since the flow security state gives no indication of the semantically matching commands. This is another area where a static security analysis is aided by a functional analysis.

4.3 Proof Example: An authentication server for a network service

An authentication server accepts a client login request, verifies that the user he/she represents has a right to the service and, if so, constructs a ticket which the client must present to the service to use it. A ticket cannot be forged. This service is similar to many distributed system applications where security is provided, [Satyanarayanan89] for example.

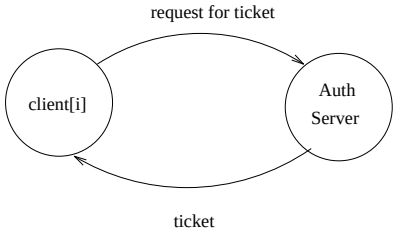

Figure 2: Authentication Service

To access the service, the client sends his name, password and his conventional encryption[3] key to the authentication server. This request is encrypted with the server's public key to dissuade network eavesdroppers and to tolerate masquerading servers since only the real server will hold the private key enabling it to decrypt the request and get the client information. The authentication server decrypts the request, authenticates the client by ensuring that the password in the message is the same as that stored on the server (in $Passwd\_Table$). The server then constructs a *ticket* which will permit the client to avail of the service provided by the main server. The ticket is encrypted with a conventional key (K_{SERVER}) known only to the server to prevent the client fabricating a ticket. The ticket also contains an *authenticator* field which contains the client's name encrypted with his key. This is to prevent some process making use of a stolen ticket. It is expected that when a client presents a ticket to the service he/she presents his key also. Since only the real client will have his key, no other process will present a key enabling the service to decrypt the authenticator to get the client name. Finally, the ticket is returned to the client encrypted with the client key. This prevents playback attacks succeeding, where a third party attempts to gain a ticket by replaying a valid request message. Only the real client will be able to decrypt the reply message.

† The ticket contains a timestamp field so that its lifetime is bounded. We don't detail the clock process. In particularly, for the proofs, we assume that the communication with *clock* effects no change to the *indirect* of the auth_server.

Security Requirements: The **server** must not be allowed to (accidentally) leak the client's ticket or personnel key to another client process. This information must be destroyed between successive calls to the authentication server. One way to guarantee this is to ensure that on each iteration of the server loop,

$$\mathcal{P} \equiv \{\ \overline{all} = \{\}, indirect = \prec \{\} \succ \ \}$$

[3]In **conventional** cryptography, the same key is used to encrypt the text and decrypt the resulting cipher-text. In **public key** cryptography, a *public* key is used to encrypt the text and a *private* key decrypts the corresponding cipher-text.

```
[ auth_server::
var client_name, client_id, client_key;
var token, authenticator, message;
var rights, date, KS_PRIV , K_SERVER;
array Rights_Table, Passwd_Table;

client_name, client_id, client_key, token, authenticator, message := 0, 0, 0, 0, 0, 0;
rights:=0; date:=0; KS_PRIV :=....; K_SERVER:=....;
Rights_Table := .... ; Passwd_Table := .... ;

*[ true →
     [ ..... □ client[i] ? message → client_name := i]; /* i is a constant */
     message := Decrypt(KS_PRIV , message);
     client_id := f^1(message); /* some function of the message */
     client_key := f^2(message);
     [ not Passwd_Table[client_name] = client_id → client[client_name] ! 0; □ /* Login failure */
       Passwd_Table[client_name]= client_id →
                                    rights := Rights_Table[client_name];
                                    clock ? date;†
                                    authenticator := Encrypt(client_key, client_name);
                                    token := f^3(authenticator, date, rights, noise_constant);
                                    token := Encrypt(K_SERVER, token);
                                    token := Encrypt(client_key, token);
                                    client[client_name] ! token; ];
     message:=0; authenticator:=0; /* Reset variables */
     token:=0; client_id:=0; client_key:=0;
     client_name:=0; date:=0; rights:=0; ];
//
client[i]::
var ticket, client_message, passwd, key, KS_PUB;
     KS_PUB := ...;
     passwd := ...;
     key := f^4(passwd);
     client_message := Encrypt(KS_PUB , (passwd, key));
     auth_server ! client_message;
     auth_server ? client_message;
     ticket := Decrypt(key, client_message);
];
```

where $\overline{all}$ represents the security variable of each variable declared in the process. Similarly, we will use $\overline{others}$ to denote the security variables not appearing in the predicate.

The **client** must not be allowed to have enough information to construct a valid ticket himself. We will show that this requirement cannot be met in theory; the client will receive a flow from the server key used to encrypt the ticket.

Outline of Server Proof: The commands preceding the loop assign constants to all the variables. It is easily shown that $\mathcal{P}$ is established as loop pre-condition. Note that for brevity we use the following annotations in the flow security predicates: *msg* for *message*, *c_msg* for *client_message*, *c_name* for *client_name*, *R_Table* for *Rights_Table* and *authent* for *authenticator*.

We insert the assertions that we want to prove into the text; assertions are enclosed in [| and |] brackets for clarity.

[| $\mathcal{P}$ |]
*[true → [| $\mathcal{R}$ |] = $\mathcal{P}$ **but** *indirect* =≺ {}{} ≻
[..... □ client[*i*] ? message → client_name := i];
[| $\mathcal{R}$ **but** { $\overline{msg}$=$\vec{msg}$ } |]
message := Decrypt(KS_{PRIV}, message);
client_id := f^1(message);
client_key := f^2(message);
[| $P_{alt} \equiv$ { $\overline{c\_name}$={i}, $\overline{c\_id} = \{msg, KS_{PRIV}, \vec{msg}\}$,
$\overline{c\_key} = \{msg, KS_{PRIV}, \vec{msg}\}, \overline{msg} = \vec{msg}$,
$\overline{others}$={}, *indirect*=≺ {}{} ≻ } |]
[**not** Passwd_Table[client_name] = client_id → client[client_name] ! 0; □
[| $R_{alt} \equiv P_{alt}$ **but** { $indirect =\prec \{P\_Table, c\_name, c\_id, msg, KS_{PRIV}, \vec{msg}\}\{\}\{\} \succ$,
$\overline{date}, \overline{authent}, \overline{rights}, \overline{token} = \{P\_Table, c\_name, c\_id, msg, KS_{PRIV}, \vec{msg}\}$ } |]
Passwd_Table[client_name] = client_id →
rights := Rights_Table[client_name];
clock ? date;
authenticator := Encrypt(client_key, client_name);
token := f^3(authenticator, date, rights, noise_constant);
token := Encrypt(K_{SERVER}, token);
token := Encrypt(client_key, token);
[| Y ≡ R_{alt} **but** $\overline{token} = \{K_{SERVER}, authent, date, rights, c\_key, c\_id,$
$msg, KS_{PRIV}, \vec{msg}, P\_Table, c\_name, R\_Table\}$,
$\overline{authent} = \{c\_key, c\_id, msg, KS_{PRIV}, \vec{msg}, P\_Table, c\_name\}$,
$\overline{date} = \{P\_Table, c\_name, c\_id, msg, KS_{PRIV}, \vec{msg}\}$,
$\overline{rights} = \{R\_Table, c\_name, c\_id, msg, KS_{PRIV}, \vec{msg}, P\_Table\}$ } |]
client[client_name] ! token; [| Y |]];
[| {Y **but** *indirect*=≺ {}{} ≻) ∨ P_{alt} |]
message:=0; authenticator:=0;
token:=0; client_id:=0; client_key:=0;
client_name:=0; date:=0; rights:=0;];
[| $\mathcal{P}$ |]

where $\vec{msg} \equiv \{KS_{PUB}, passwd, key, c\_msg\}$

$\mathcal{R}$ follows from $\mathcal{P}$ using $R_{s-repetitive}$. The body of the first alternative is a simple assignment. $A_{s-receive}$ allows any predicate to be established; since the guards are not Boolean expressions, no change to *indirect* occurs on termination. P_{alt} follows from the subsequent assignments using $A_{:=_s}$ and $R_{s-composition}$. Each branch of the alternative will start in condition R_{alt} which follows from P_{alt} using $R_{s-alternative}$. In the second of the branches, $A_{:=_s}$ is used to show that Y is established before the communication with the client. Since A_{s-send} permits any post-condition, we choose Y. Termination of the alternative gives (Y **but** $indirect =\prec \{\}\{\} \succ) \vee P_{alt}$ using $R_{s-alternative}$. The P_{alt} is established if the first branch executes. Lastly, it can be easily shown that the final assignments of the loop re-establishes $\mathcal{P}$ from the alternative post-condition.

Thus, assuming that the proof of the server co-operates with the proof of the other processes, the predicate $\{\mathcal{P}\}$ holds each time a client request is received. Therefore, we know that the server is unable to retain valuable client information after servicing a client.

Outline of Client Proof

passwd := .... ;
KS_{PUB} := ...;
[| { $\overline{all} = \{\}$, $indirect = \prec \{\} \succ$ } |]
key := f^4(passwd);
client_message := Encrypt(KS_{PUB}, passwd, key);
$\mathcal{N} \equiv$ [| {$\overline{c\_msg} = \{KS_{PUB}, passwd, key, c\_msg\}$,
$\overline{key} = \{passwd\}$, $\overline{others}$={}, $indirect$=$\prec$ {} $\succ$ } |]
auth_server ! client_message;
[| $\mathcal{N}$ |]
auth_server ? client_message;
[| {$\overline{c\_msg} = \vec{c\_msg}$, $\overline{key} = \{passwd\}$, $\overline{others}$={}, $indirect$=$\prec$ {α} $\succ$ } |]
ticket := Decrypt(key, client_message);
[| {$\overline{ticket} = \{key, c\_msg, passwd, \vec{c\_msg}\}$, $\overline{c\_msg} = \vec{c\_msg}$, $\overline{key} = \{passwd\}$,
$\overline{others}$={}, $indirect$=$\prec$ {α} $\succ$ } |]

where $\vec{c\_msg} = \alpha \vee \beta$
$\beta \equiv \{K_{SERVER}, authent, date, rights, c\_key, c\_id, msg, KS_{PRIV}, token,$
$KS_{PUB}, passwd, c\_msg, P\_Table, c\_name, R\_Table\}$
$\alpha \equiv \{P\_Table, c\_name, c\_id, msg, KS_{PRIV}, KS_{PUB}, passwd, key, c\_msg\}$

The pre-condition to the send to server communication is established from the process pre-condition. The communication axioms allow any post-condition

so, in the example above, we just need to show that the process proof co-operates with the server to prove the post-condition and thus the client.

The result for the client (the state of the *tickets* security variable in the post-condition) means that we cannot in theory prove our system secure with respect to the client; he/she receives an information flow from several sensitive variables - the password and rights tables, the server's private key and the server's personnel key used to encrypt the tickets. But things are not quite as bad as they seem: despite the flow from the password and rights table, a combined functional proof would show that the information is from the entry corresponding to that client only[4]. The information flow from the server's private key arises because it is used for encrypting the ticket. This flow cannot be avoided. The security of the system will thus depend on the strength of the encryption algorithm. This is precisely what happens in practice, so the onus is now on the security officer to satisfy himself with the strength of the encryption algorithms.

Co-operation needs to be established for the three communications: the client to server, the server to client error message and the server to client result. We will show the latter using $A_{communication}$:

$z_{server} = indirect_{server} =$

$$\prec \{P\_Table, c\_name, c\_id, msg, KS_{PRIV}, KS_{PUB}, passwd, key, c\_msg\}\{\}\{\} \succ$$

$z_{client[client\_name]} = indirect_{client[client\_name]} = \prec \{\} \succ$

$A_{communication}$ says that after rendezvous

$indirect_{server} = z_{server} \sqcup z_{client[client\_name]}$

but since z_{client} is $\prec \{\} \succ$

$\equiv z_{server} \equiv indirect_{server}$ which is established in the server. (See predicate Y).

$indirect_{client[client\_name]} = (\{c\_name\} \cup \overline{c\_name}) \uplus (z_{client[client\_name]} \sqcup z_{server})$

$\equiv \prec \{P\_Table, c\_name, c\_id, msg, KS_{PRIV}, KS_{PUB}, passwd, key, c\_msg\} \succ$

This security variable expression is equal to α that we defined in the proof of the client process.

Finally, also from the communication axiom, we consider the update to the $client\_message$ variable.

$\overline{c\_msg} = \{token\} \cup \overline{token} \cup val(indirect_{server}) \cup val(indirect_{client[client\_name]})$

$\equiv \{token, K_{SERVER}, authent, date, rights, c\_key, c\_id, msg, KS_{PRIV},$

$KS_{PUB}, c\_msg, passwd, P\_Table, c\_name, R\_Table\}$

which is met since it is equivalent to β. The other two communications can also be shown to be consistent with the communication axiom. □

[4] A dynamic mechanism can more easily support a flow variable per array entry

5 Adaptability of the Flow Control mechanism

The flow mechanism up to now has been based on security variables: each variable is tagged with the set of variables from which it has received a flow. A proof system was described for verifying the flow security of a programme. The section looks at the generality of this mechanism from three points of view. We firstly more formally compare the security variable and traditional security level, or classification, approaches to flow control. We then look at the problems encountered with a dynamic or runtime version of the flow mechanism. Finally, we see how our mechanism can be adapted to support coarser grained policies - flow policies where permitted flows are specified in terms of what processes or groups of processes may legally exchange data rather than what variables may do so.

5.1 Information flow control: security variables versus security levels

Information flow in programmes is traditionally handled by giving each variable a security level, or classification. The set of classifications forms a lattice[5] so that the effect of the assignment $y := exp(x_1, x_2, ..., x_n)$ in the case where variables are dynamically bound to classifications, is to make the security classification of variable y, denoted $\underline{y}$, the least upper bound of the right hand side variables' classifications. In programmes where security classifications are statically bound to variables, such an assignment is illegal if the resulting classification exceeds that of y.

Theorem ————————————————————————————————-
The relationship between the security variable approach and the classification approach is captured by the following predicate:

$$x \in \overline{y} \Rightarrow \underline{x} \leq \underline{y}$$

where $\underline{x}$ is the classification of x when the information it contained was transmitted to y. $\diamond$
——-

[5] A lattice is a partially ordered set of elements S on which some relation, denoted $\leq$, is defined. The partial ordering implies *anti-symmetry* - $\forall a, b \in S, a \leq b \wedge b \leq a \Rightarrow a = b$, *reflexivity* - $\forall a \in S, a \leq a$ and *transitivity* - $a \leq b \wedge b \leq c \Rightarrow a \leq c$. In addition, for all pairs of elements a, b there is an element known as the *least upper bound* $(a \oplus b)$ and an element known as the *greatest lower bound* $(a \otimes b)$ defined as follows:
$a \oplus b \equiv c$ if $a, b \leq c \wedge \neg\exists d, a, b \leq d, d \leq c$ and $a \otimes b \equiv c$ if $a, b \geq c \wedge \neg\exists d, a, b \geq d, d \geq c$

Proof We will prove the theorem with respect to the assignment operation:

$$y := \exp(x_1, x_2, ..., x_N)$$

The proof is by induction on the contents of the security variables. That is, we show that our property is true at step(0) - at the start of the programme and that it is preserved by the initial assignment (the base step). We then show that if the property holds at step(N), then it holds at step(N+1). Step(n) denotes the flow security state after execution of the n^{th} instruction.

After termination of the above assignment, with respect to the security classifications, where the r_is are the variables which flow indirectly in the assignment and $\oplus$ is the least upper bound operator[5], we have

$$\underline{y} \geq (\oplus_{i=1}^{N} x_i) \oplus (\oplus_{i=1}^{M} r_i) \qquad\qquad c\_property$$

since, for dynamic binding, the classification of y is set to the least upper bound of the right hand side variables while for static binding, the command will only terminate (without an error) if the property holds. The $\oplus$ operator allows us to re-write the *c_property* as $\forall i = 1..N, \underline{y} \geq \underline{x_i} \wedge \forall i = 1..M, \underline{y} \geq \underline{r_i}$

The security variable flow mechanism gives

$$\overline{y} = \{x_i \mid i = 1..N\} \cup \{\overline{x_i} \mid i = 1..N\} \cup \text{val}(indirect)$$
$$sv\_property$$

Base Step At the start of the programme, the security variables and *indirect* will be empty. Each variable will have some initial classification. The assignment $y := \exp(x_i)_{i=1..N}$ would give $\overline{y} = \{x_i\}_{i=1..N}$. Using *c_property*, the theorem is trivially satisfied for this case.

Inductive Step We assume that the property holds at the n^{th} step.

$$\forall\ e_j \in \overline{x_i} \Rightarrow \underline{e_j} \leq \underline{x_i}$$

where $\underline{e_j}$ is the classification of the information in e_j when it was transmitted to x_i. We show that for all the entries in $\overline{y}$, the theorem holds. We take each term of *sv_property* in turn:

$\underline{\{x_i\}_{i=1..N}}$

For all x_i, *c_property* tells us that $\underline{y} \geq \underline{x_i}$. Thus the theorem holds for the x_is.

$\underline{\{\overline{x_i}\}_{i=1..N}}$

The **inductive step** assumption tells us that $\underline{x_i} \geq \underline{e_j}$ for all $e_j \in \overline{x_i}$. Thus, by *c_property* and the transitivity of the $\geq$ operator,

$$\underline{y} \geq \underline{x_i} \wedge \underline{x_i} \geq \underline{e_j} \Rightarrow \underline{y} \geq \underline{e_j}$$

so the theorem also holds for the $\{\overline{x_i}\}_{i=1..N}$ terms. Note that an e_j may be the same variable as one of the x_is, this does not matter.

$\underline{\text{val}(indirect)}$

This variable is of the form $\{r_i \mid i = 1..M\}$. The proof that the theorem holds for this follows directly from *c_property*.

Since the theorem holds for the inductive and the base steps, it is proven. The proof for the other programme constructs easily follows. □

Note how the theorem stated for $\underline{x}$ "when the information it contained was transmitted". This is because a variable named in a security variable may subsequently have its classification altered by an independent flow. For statically bound classifications, this cannot happen; the class will always be constant. This gives us the following corollary.

Corollary 1 For variables x, y whose classifications are statically bound, the predicate:

$$x \in \overline{y} \Rightarrow \underline{x} \leq \underline{y}$$

holds at all times. ◇

The reverse implication of corollary 1, $\underline{x} \leq \underline{y} \Rightarrow x \in \overline{y}$, does not hold. A flow mechanism using security levels is unable to say anything about information flows between particular variables. In this sense, the traditional security level approach is "poorer" than the security variable approach.

5.2 Run-time and Compile-time Detection of Flow Violations

In the preceding sections, we introduced the security semantics of a parallel programming language in an axiomatic form and used the semantics to verify, by hand, the security of a programme. But the language semantics give us a framework for other forms of security violation detection, principally, at run-time and compile-time.

Run-time: For an executing programme, we need some way of expressing our flow security policy since the mechanism (unlike the security classification approach) has no implicit policy. The latter means that we must also specify the precise points in a programme where the policies are to hold. In the authentication server example, the server's security constraints only had to be met at the start of each iteration. We could not have done this with the traditional classification mechanism.

We introduce a pragma declaration, **ENSURE**, which specifies and verifies at run-time the flow security policy. Its syntax is the following:

$$\textbf{ENSURE } \Gamma_v \textbf{ not in } \overline{v}$$

where Γ_v is the set of variables forbidden to flow to v under the security policy. The semantics of the pragma are the following:

A_{ENSURE}

$$\frac{\mathrm{P}.v \cap \Gamma_v = \emptyset}{\{\mathrm{P}\}\ \textbf{ENSURE } \Gamma_v \textbf{ not in } \overline{v}\ \{\mathrm{P}\}}$$

If the command is executed in a state satisfying P where the security variable for v is disjoint from the set of variables forbidden to flow to it, then the command terminates normally, still in flow security state satisfying P. The semantics are undefined in the event of a security violation.

One of the inherent problems of a dynamic mechanism is that failures can leak information to the environment - this information being the condition that led to the failure. Consider the following piece of code:

$$[\ b = 7 \rightarrow c := d \ \Box\ b \neq 7 \rightarrow \textbf{skip}\ ]$$
$$\textbf{ENSURE } \{\mathrm{d}\} \textbf{ not in } \overline{c}$$

The system policy may forbid flows from variable d to c. During execution, should the condition $b = 7$ be true, then the policy will be violated and the programme fails. Since a failure only occurs when the value of the condition is true, the value of variable b is leaked to the environment. Note that our proof system would cause this programme to be rejected because of the possible flow from d to c. Another example of information being leaked to the environment is when a process terminates because one of its alternative commands fails. Our proof system deals only with inter variable flows; it cannot cope with this problem since the failure event is not deducible from other process' variables.

The run-time mechanism must tackle the problem by considering the environment as part of the system. Each time a process' *indirect* is updated, an error is signaled if information concerning the updated variables are forbidden to flow to the environment. In the preceding example, an error is raised if the value of b cannot be released to the environment of the process. In the case of the alternative command, when *indirect* is updated on entry in **update1**, we check that it contains no variables which can be leaked to the environment by the command failing. In effect, what is happening is that we are unconditionalising process violations. However, there is a snag with this approach - the value of *indirect* cannot be undone after a conditional command as is usually the case. To see why, consider the following example: Process P2 fails if the condition $x = 0$ is true in P1. Yet x will not be in P2's

```
[
  P1::                          P2::
  [                             [
  ⋮                             ⋮
  [ x=0 → y:=a □                P1 ? b
    x≠0 → y:=1 ];               ENSURE {a} not in b
  P2 ! y
  ⋮                             ⋮
  ]                             ]
  //
]
```

indirect after the rendezvous if P1's *indirect* is undone after the alternative. A dynamic mechanism must not undo the *indirect*s.

The optimum solution would be to have flow violations detected by the compiler. With violations statically detected, we do not have the problems with *indirect* that existed for the run-time approach. Some aspects of the compilation approach are studied in [Mizuno & Oldehöeft.88] and [Mizuno & Schmidt.92].

5.3 Supporting coarser grained flow policies

A feature of the security variable based mechanism is the size of the security variables becoming very large. For information flow policies expressed in terms of the processes that may exchange information rather than the variables in those processes, our proof system gives a lot of supplementary information. For example, the stock exchange has the secrecy rule that traders

dealing with one company must not receive "insider" information concerning another. Computerised versions of the stock exchange would specify that no flows may occur from accounting processes with a company X attribute to accounting processes with other company attributes [Brewer & Nash.89]. Our flow mechanism can be easily tailored to meet these criteria.

To verify process-based policies, we re-write the proof system. The first thing that needs to be done is to have variables named in the security variables pre-fixed by the name of the process in which they were declared. (This should have been done in the first place to avoid name clashs). For example, $\overline{x} = \{\text{P1.s, P2.t}\}$ means that x has received a flow from variable s of process P1 and variable t of process P2. Thus, the assignment $y := \exp(x_i)_{i=1..N}$ would set:

$$\overline{y} = \{self.x_i \mid i = 1..N\} \cup \{self.\overline{x_i} \mid i = 1..N\} \cup \text{val}(indirect_{self})$$

where *self* names the enclosing process. The security variable for a constant is still the empty set {}. The "assignments" treating the indirect flows in the conditional commands are similarly handled. Thus, in $R_{repetitive}$,

$$\overline{l} = self.\overline{l} \cup \{self.c\} \cup self.\overline{c}$$

for all variables c in the Boolean guards and all variables l in the lhs_vars. The communication axiom has as part of its post-condition for $P_R!x//P_S?y$

$$\overline{y} = \{P_S.x\} \cup \{P_S.\overline{x}\} \cup \text{val}(indirect_S) \cup \text{val}(indirect_R)$$

For policies where only the process name is important, then one need only store the process name in the security variable. Thus, the assignment and communication predicates above would be respectively:

$$\overline{y} = \{self\} \cup \{self.\overline{x_i} \mid i = 1..N\} \cup \text{val}(indirect_{self})$$
$$\overline{y} = \{P_S\} \cup \{P_S.\overline{x}\} \cup \text{val}(indirect_S) \cup \text{val}(indirect_R)$$

One can go a step further than this. Policies on processes may be based on attributes of the processes rather than the processes themselves. In the Chinese Wall example, the attributes of interest of the process are the company name for which the process is working. The system maintains a mapping from each process to the attributes associated with that process. One could imagine process names in the security variables being replaced by their attributes. Thus one could know the attributes of the information flow sources. The assignment and communication axioms are as in the last paragraph except that *self* and the process name now stand for the attributes that the system associates with the process in question.

Examples Consider how a multi-level secure environment might be implemented. Each process is given a classification in the set $\{c_1,....,c_N\}$ on which there is a lattice ordering ($\leq$). We will assume here that the classification of a process is static. The security variables contain elements of the set $\{c_1,....,c_N\}$. The classification of the information that has flown to each variable is the least upper bound or maximum of the security variable entries. Therefore, to ensure multi-level secrecy in a process of classification c_i, we insert the predicate for variable v:

$$\{c_j \mid self < c_j \leq c_N \} \cap \overline{v} = \emptyset$$

at the points in the programme where the policy is meant to hold. In a similar way, if the set of company names in a Chinese Wall applications is denoted by the set COMPANIES, the predicate to be proved for variable v would be:

$$(\text{COMPANIES} - \{self\}) \cap \overline{v} = \emptyset$$

6 Discussion

In this paper, an information flow mechanism for a parallel language was presented. There are two main arguments. Firstly, the flow semantics based on security variables rather than the traditional classification scheme gives us more flexibility in our security analysis. Secondly, a weaker form of indirect information flow consistency can be used to capture inter-process indirect flows so that no extra message exchanges are necessary. A set of axioms and rules for establishing the information flow security of a CSP programme was presented.

6.1 Related Work

6.1.1 Language Approach

[Denning75] was the first to consider information flow control in programmes. Using the classification approach, she describes a compile-time algorithm for ensuring that sequential programmes with statically bound security classes are flow-correct - that no variable receives information whose classification is greater than that of the variable. In programmes where the security classifications are dynamically bound, a runtime mechanism is introduced so that the class of y is updated with the value of the flow. See also [Denning76], [Denning & Denning.77] and [Denning82].

[Reitman78] describes a flow control proof system for parallel processes communicating by shared variables as well as by message passing using the

security classification approach. A proof system for Parallel Pascal is also presented. His assignment axiom is:

$A_{:=}$

$$\{\ P[\underline{y} \leftarrow \underline{x_1} \oplus \underline{x_2} \oplus .... \oplus \underline{x_n} \oplus local \oplus global]\ \}$$
$$y := exp(x_1, x_2, ..., x_n)$$
$$\{\ P\ \}$$

where the variables *local* and *global* fulfil the same role as *indirect*. Shared variable communication is treated as an extension to sequential processing. To prove a set of Pascal processes, for each process he establishes an invariant - a definition of the maximum security level of each of the variables, and then for the other processes, shows that when they communicate with the process, the invariant in the called process cannot be violated. See also [Andrews & Reitman.80].

However, Reitman's flow semantics for the conditional commands are incomplete - he does not consider the indirect flow from the selection guard to the branch which is not executed. His selection command rule **if** B **then** $S1$ **else** $S2$ **endif**, is the following,

$$\frac{\{\ V,L',G\ \}\ S_i\ \{\ V,L',G'\ \},\ i=1,2 \qquad (V,L',G) \Rightarrow L[local \leftarrow local \oplus \underline{B}]}{\{\ V,L,G\ \}\ \textbf{if}\ B\ \textbf{then}\ S1\ \textbf{else}\ S2\ \textbf{endif}\ \{\ V,L,G'\ \}}$$

{ V,L,G } is a predicate on the security state, V a predicate on the mapping from variables to classifications, L a predicate on the value of *local* and G a predicate on the value of *global*. In each branch, the value of *local* which captures the indirect flow from the programme guard is updated on entry: L goes to L'. Consider the following programme segment where a and b are known to be 0 beforehand:

$$\{\ \underline{a} = \underline{b} = low,\ local = low,\ global = low\ \}$$
$$\textbf{if}\ x = 0\ \textbf{then}\ a := 1\ \textbf{else}\ b := 1\ \textbf{endif}$$
$$\{\underline{a} \leq \underline{x},\ \underline{b} = low,\ local = low,\ global = low\} \vee \{\underline{a} = low,\ \underline{b} \leq \underline{x},\ local = low,\ global = low\}$$

And since the post-condition allows us to conclude $\{\underline{a} \leq \underline{x},\ \underline{b} \leq \underline{x}\}$, the indirect flow in the two branches seems to be accounted for. But it is not! The logical or in the post-condition is in fact an exclusive-or. The predicate which holds depends on the branch which executes which is exclusive. This can be seen from an example used in [Andrews & Reitman.80] in which they give the following piece of code:

$$\textbf{if}\ b\ \textbf{then}\ y := x\ \textbf{endif}$$
$$\textbf{if}\ \neg b\ \textbf{then}\ z := y\ \textbf{endif}$$

b is a Boolean expression not containing y say, both $\underline{z}$ and $\underline{y}$ are *low* beforehand. Reitman correctly points out that since only one branch can execute, there is no flow from x to z. Using his combined functional and security proofs he establishes the post-condition:

$$\{\ b \Rightarrow (\underline{y} \leq \underline{b} \oplus \underline{x}, \underline{z} = low), \neg b \Rightarrow (\underline{y} = low, \underline{z} \leq \underline{b} \oplus \underline{x})\ \}$$

This clearly does not cater for all indirect flows since in the first case z does not have the flow from b registered, y in the second. The problem is that his OR in the selection semantics is not logically strong enough to capture all indirect flows - as we saw it can behave as an exclusive-or. This has profounder implications for parallel programmes where we must register indirect flows in other processes. Consequently, Reitman's semantics can allow an insecure programme to be "proved" secure.

[Mizuno & Oldehöeft.88] also use the security classification approach in the context of a distributed object oriented system. All inter-process (object) indirect flows are considered. Each time a communication is skipped in a conditional command a *probe*, or dummy message, is sent to each object which could possibly have received a flow if the method call had been made. The probe carries the security level value of the condition on which the method call was skipped which is least upper bound with the class of the variables in the receiving objects which might have received a direct flow. However, as argued in the text, the sending of such messages is not needed.

The framework in which the language approach to information flow began was multi-level security. In multi-level security, each process has a fixed level (where the set of levels form a lattice). A process can read an object only if the object's level is not greater than that of the process; a process can write to an object only if the object's level is not less than that of the process. Thus if a process (programme) has a sensitivity level i, its programme variables can contain information from different levels at the same time. If a programme variable v held information from level $i-2$ then the process should be allowed to write v to an object of level $i-1$ even though the security level of the process would normally forbid this. For this to be allowed, a mechanism for controlling information flows at the programme level must be incorporated.

Information flow control where each variable has a security level supposes an environment where there is a multi-level-secure policy. This is too inflexible with respect to the range of policies that one might like to support. Another problem with the security level approach is deciding whether some variable should be statically or dynamically bound to its level. Static binding is very restrictive in that the sensitivity of the variable is independent of the

information it contains; after reseting a variable, its level would not be *Low*. Such over-classified variables lead to needless flow violations being signaled. Dynamic binding has the problem that the classification is too dependent on the information contained in the variable; for example, the constant assignment in a bank account process *balance* := 50.000FF would set $\underline{balance}$ to *Low* which would mean that *balance* could legally flow to any variable. Finally, and most importantly, the classification approach has no way of saying that a variable y depends on some variable x. With security variables, as long as the value of y depends on the (perhaps former) value of x then $x \in \overline{y}$. This is the property that our flow control mechanism captures and is far more useful than the policy-implicit classification scheme.

[Jones & Lipton.75] describe a surveillance set flow mechanism for sequential programmes which also accumulated variables in sets. The mechanism presented in this paper differs in how indirect flows are handled, parallelism is addressed and that efficiency and correctness considerations are looked at.

6.1.2 Multi-level Secure systems

Despite the work of Denning and Reitman, the approach more often used for multi-level security is to assign a security level to each process [Casey et al.88], or site as in Unix United [Randell & Rushby.83], and to prevent messages being sent from processes to higher level processes. Control of message exchanges is easier and less costly than controlling the flows at the language level.

6.1.3 Commercial Security

Another, though non-multi-level approach to information flow control is the Chinese Wall model [Brewer & Nash.89]. Here, a process' permitted access to the objects in the system is defined as a function of the system objects that it has already accessed. The objects of the system are partitioned into *data-sets* which are themselves partitioned into conflict of interest classes. The rule is that a process may only access an object if it has not already accessed an object in another data-set of the same conflict of interest class. The Chinese wall is a model of the stock market security policy.

6.1.4 Derivation of Secure Systems

A more recent approach integrates security validation into the system design. This consists of analysing the system's specification to try and find flows between the input and outputs. The specification should be much easier to analyse than a complete system. The system is then derived by refining the

secure specification. The various flavours of this approach - *non-interference*, *non-deducibility* and now *causality* are well summarised in [Bieber & Cuppens.92]. More specifically, the system is viewed as a state machine which must satisfy certain security properties. For example, the non-interference property can be informally stated as "a user A is non-interfering with a user B if a state transition caused by A leads to a state which is equivalent to the former state from B's point of view". This is a top down approach. The specification is refined and at each step it must be shown that the property still holds. These approaches have the advantage that they are less pessimistic for flows since some of the dependencies noted in our mechanism are too small to signal any useful information. However, they do have their difficulties. When subsystems that satisfy non-interference are joined together, the resulting system need not be non-interfering. Non-interference requires that system specifications be deterministic, otherwise covert channels can appear during refinement.

A related approach is taken by [Jacob88,90] and [Foley87] who use the trace model of the 1985 version of CSP [Hoare85] to reason about security properties of a system's specification.

6.2 Other Aspects

The flows considered in this paper are those that occur between variables in the programme. There are other channels of information flow. In particularly, programme execution time is a way of leaking information to the environment [Lampson73]. For the command:

$$*[\ x = 0 \rightarrow \mathbf{skip}\ ]$$

an observer of the process can easily discern that the value of x is zero if after some time the programme has not terminated. [Reitman78] did not consider such timing channels since he just wanted to capture how variables influenced each other. [Jones & Lipton.75] point out that timing channels are created by the variable execution paths of conditional commands and the information signaled comes from the command guard variables. One could thus imagine a pseudo-variable *timing* inserted into each conditional command which signals an error each time it receives an indirect flow from a variable that is not allowed to influence the execution time of the programme. However, timing channels are more difficult to interpret in a real concurrent computing environment since execution time depends on the number and computational nature of currently executing processes.

An interesting question to pose at the end of the paper is how the information flow semantics presented may be adapted to other models of process communication. Evidently, the flow semantics are very dependent on the kind of communication used. For processes communicating asynchronously for example, the sender cannot know at communication time whether the message was consumed by the receiver. Thus, there is only a transfer of *indirect* from the sender to receiver at communication, the reverse does not occur. Another kind of communication uses letter boxes. In such systems the receiver may not always know the sender. It is also possible that messages are consumed by the receiver in an order different to their arrival. All of these factors have important consequences for the information flow analysis. Thus, a proper flow analysis of any programming model and its security semantics requires a complete understanding of the language model behaviour. Nevertheless, we believe the approach taken here to be general enough to serve as a basis for undertaking the design of a security proof system for different parallel programming paradigms.

Acknowledgements The authors are extremely grateful to Daniel Le Métayer and Valérie Issarny and Claude Jard for their careful readings of this paper.

A condensed version of this paper appears in the Proceedings of the 6^{th} Workshop on the Foundations of Computer Security. The paper is entitled "Information Flow Control in a Parallel Language Framework".

Bibliography

[Andrews & Reitman.80]

Andrews (G.R.), Reitman (R.P.), "An Axiomatic Approach to Information Flow in Programs", in *ACM Transactions on Programming Languages and Systems*, volume 2 (1), January 1980, pages 504-513.

[Apt et al.80]

Apt (K.R.), Francez (N.), De Roever (W.P.), "A Proof System for Communicating Sequential Processes", in *ACM Transactions on Programming Languages and Systems*, volume 2 (3), July 1980, pages 359-385.

[Apt 83]

Apt (K.R.), "Formal Justification of a Proof System for Communicating Sequential Processes", in *Journal of the ACM*, volume 30 (1), January 1983, pages 197-216.

[Bieber & Cuppens.92]

Bieber (P.), Cuppens (F.), "A Logical View of Secure Dependencies", in *Journal of Computer Security*, volume 1 (1), Summer 1992, pages 99-129.

[Brewer & Nash.89]

Brewer (D.), Nash (M.), "The Chinese Wall Security Policy", in *Proceedings of the IEEE Symposium on Security and Privacy*, 1989, pages 206-214.

[Casey et al.88]

Casey (T.), Vinter (S.), Weber (D.), Varadarajan (R.), Rosenthal (D.), "A Secure Distributed Operating System", in *Proceedings of the IEEE Symposium on Security and Privacy*, 1988, pages 27-38.

[Cohen77]

Cohen (E.), "Information Transmission in Computational Systems", in *Proceedings of the Sixth ACM Symposium on Operating System Principles*, 1977, pages 133-139.

[Denning75]

Denning (D.E.), *Secure Information Flow in Computer Systems*, Phd Thesis, Purdue University, May 1975.

[Denning76]

Denning (D.E.), "A Lattice Model for Secure Information Flow", in *Communications of the ACM*, volume 19 (5), May 1976, pages 236-243.

[Denning & Denning.77]

Denning (D.E.), Denning (P.J.), "Certification of Programs for Secure Information Flow", in *Communications of the ACM*, volume 20 (7), July 1977, pages 504-513.

[Denning82]

Denning (D.E.), *Cryptography and Data Security*, Addison-Wesley Publications, 1982.

[Foley87]

Foley (S.N.), "A Universal Theory of Information Flow", in *Proceedings of the 1987 Symposium on Security and Privacy*, Oakland, California, 1987, pages 116-122.

[Hoare69]

Hoare (C.A.R.), "An Axiomatic Basis for Computer Programming", in *Communications of the ACM*, volume 12 (10), October 1969, pages 576-583.

[Hoare78]

Hoare (C.A.R.), "Communicating Sequential Processes", in *Communications of the ACM*, volume 21 (8), August 1978, pages 666-674.

[Hoare85]

Hoare (C.A.R.), *Communicating Sequential Processes*, Prentice-Hall Londen, 1985.

[Jacob88,90]

Jacob (J.L.), "Security Specifications", in *Proceedings of the 1988 Symposium on Security and Privacy*, Oakland, California, 1988, pages 14-23. A reduced version of this paper can be found in *Developments in Concurrency and Communication*, chapter 7, editor C.A.R. Hoare, Addison-Wesley Publishing Company, 1990.

[Jones & Lipton.75]

Jones (A.), Lipton (R.), "The Enforcement of Security Policies for Computations", in the *Proceedings of the* 5^{th} *Symposium on Operating System Principles*, November 1975, pages 197-206.

[Lampson71,74]

Lampson (B.), "Protection", in *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, Princeton University, March 1971, pages 437-443, reprinted in *Operating Systems Review* 8,1, January 1974, pages 18-24.

[Lampson73]

Lampson (B.), "A Note on the Confinement Problem", in *Communications of the ACM*, volume 16 (10), October 1973, pages 613-615.

[Levy84]

Levy (H.), *Capability-based Computer Systems*, Digital Press Publications, 1984.

[Mizuno & Oldehöeft.88]

Mizuno (M.), Oldehoeft (A.), *Information Flow Control in a Distributed Object-Oriented System: Parts I & 2*, Kansas State University, Report TR-CS-88-09. May 1988.

[Mizuno & Schmidt.92]

Mizuno (M.), Schmidt (D.), "A Security Control Flow Control Algorithm and Its Denotational Semantics Correctness Proof", *Journal on the Formal Aspects of Computing*, 4 (6A), November 1992, pages 722-754.

[Neuman91]

Neuman (B.C.), "Protection and Security Issues for Future Systems", in *Operating Systems of the 90s and Beyond*, Proceedings published in *Lecture Notes in Computer Science*, volume 563, pages 184-201.

[Plotkin83]

Plotkin (A.), "An Operational Semantics of CSP", in D. Björner, editor, *Formal Description of Programming Concepts - II*, North Holland Publishing Company, pages 199-225, 1983.

[Randell & Rushby.83]

Randell (B.), Rushby (J.), "A Distributed Secure System", in *IEEE Computer*, volume 16 (7), July 1983, pages 55-67.

[Reitman78]

Reitman (R.), *Information Flow in Parallel Programs: an Axiomatic Approach*, Phd Thesis, Cornell University, August 1978.

[Saltzer & Schroeder.75]

Saltzer (J.), Schroeder (M.), "The Protection of Information in Computer Systems", in *Proceedings of the IEEE*, volume 63 (9), September 1975, pages 1278-1308.

[Satyanarayanan89]

Satyanarayanan (M.), "Integrating Security in a Large Distributed System", in *ACM Transactions on Computer Systems*, volume 7 (3), August 1989, pages 247-280.

Appendix - Formal Justification of the Security Proof System

There are two stages in the justification (see figure 3). Firstly, we must show that the flow semantics $\mathcal{M}_S$ capture all the flows that arise, that is, that the modifications to the flow security state made by each command are *necessary* and *sufficient* to capture the flows of information. Secondly, we show that the axioms and rules of the proof system are *sound* and *complete* with respect to $\mathcal{M}_S$.

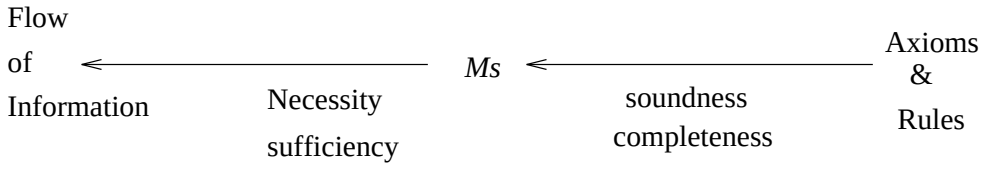

Figure 3: Plan of Proof

Notation

A formula is of the form {p} S {q} [Hoare69], where p and q are predicates on the flow security state and S a programme segment. We let $\models$ {p} S {q} denote a formula that is true given our flow semantics $\mathcal{M}_S$. That is, if predicate p holds before S executes, then if and when the programme terminates, predicate q holds on the security state. $\vdash$ {p} S {q} means that our proof system allows us to show that q follows from executing S in a state satisfying p. If a predicate p is true in some flow security state σ, then we write $\models p(\sigma)$.

Aside We make no reference to the interpretation J in the following presentation, to avoid any ambiguity we assume it to be standard; for example {a} $\cup$ {b,c} = {a,b,c}, true $\vee$ false = true, etc. $\diamond$

The "+" operator "updates" a state σ,

$$(\sigma + (x \rightarrow E))(v) = \begin{cases} \sigma(v) & \text{if } x \neq v \\ E & \text{if } x = v \end{cases}$$

x can either be a security variable or the variable *indirect*.

When discussing parallelism, we write $\vec{S}$ for $S_1 \parallel ... S_i ... \parallel S_N$.

A Formal justification of the flow security semantics

For some given parallel sub-programme $\vec{S}$, suppose we have

$$(\vec{S}, \sigma) \rightarrow^* (\vec{\epsilon}, \tau) \text{ and } x \notin \sigma(\overline{y})$$

where σ and τ are combined functional and flow security states. The sequence of transitions form a trace denoted $\mathcal{T}$. We consider in turn what it means for x to be a member (resp. not a member) of $\overline{y}$ in state τ.

$x \in \tau(\overline{y})$: If this is true then there must exist some value for x, different from $\sigma(x)$ such that one cannot form a trace $\mathcal{T}$' with the same path of control as $\mathcal{T}$, and even if one can, the final value for y may differ. That this property holds shows the necessity of our flow semantics.

$x \notin \tau(\overline{y})$: In this case, y does not reveal any information concerning x. This should mean that no matter what the initial value of x in σ, there is a trace $\mathcal{T}$', formed by $(\vec{S}, \sigma') \rightarrow^* (\vec{\epsilon}, \tau')$ where $\sigma' = \sigma + (x \not\rightarrow \sigma(x))$ such that $\mathcal{T}$' has the same control path as $\mathcal{T}$ and $\tau(y) = \tau'(y)$. That this property holds shows the sufficiency of our flow semantics.

A.1 Sufficiency

We denote the property of sufficiency on a sub-programme S as Γ(S).

The proof is by induction. We assume that for any parallel sub programme, $\vec{S'}$, that $\Gamma(\vec{S'})$ is true. We proceed to prove that $\Gamma(\vec{S})$ where $\vec{S}$ is $\vec{S'}$ extended by C = **skip**, $y := E(v_i)$, [i = 1..N $\Box C_i \rightarrow S_i$;] and *[i = 1..N $\Box C_i \rightarrow S_i$;] and finally by the communication command. The base step, proof of Γ(C) proves to be a generalisation of the inductive step. Throughout we rely on the following **composition property**:

If variety in an input v of command S1 leads to variety in an output x, and if variety in the input x of command S2 leads to variety in the output y, then variety in the input v of S1;S2 leads to variety in the output y.$\diamond$

C = skip

Given $\mathcal{M}$ is the functional CSP semantics [Plotkin83], we have from $\mathcal{M}$[**skip**], (**skip**, σ) $\rightarrow$ (ϵ, σ) and (**skip**, σ') $\rightarrow$ (ϵ, σ'), that is τ equals σ and τ' equals σ', where σ, σ', τ and τ' are defined above in the proof obligation.

Base case No state changes occur; τ(y) = τ'(y) trivially since σ(y) equals σ'(y).

Inductive step Since **skip** alters nothing, Γ(S;**skip**) if and only if Γ(S). But this is guaranteed from the induction hypothesis.

C $= y := E(v_i)$i=1..N

The semantics $\mathcal{M}$ of the assignment command gives

$$(y := E(v_i)\text{i=1..N}, \sigma) \rightarrow (\epsilon, \sigma + (y \rightarrow E(\sigma(v_i))\text{i=1..N})$$
$$(y := E(v_i)\text{i=1..N}, \sigma') \rightarrow (\epsilon, \sigma' + (y \rightarrow E(\sigma'(v_i))\text{i=1..N})$$

Base case Our flow semantics tell us that if $x \notin \overline{y}$ in τ at the finish, then x is not one of the variables v_i, on the right hand side of the := operator. (In the initial state *indirect* and all security variables are empty). We presume that only those variables named in the expression may influence the left hand side variable. So varying x in the initial state σ' makes no difference on y, that is $\tau(y) = \tau'(y)$

Inductive step $x \notin \overline{y}$ after termination implies from $\mathcal{M}_S$ that x is not on the right hand side of ":=" i.e. one of the v_i variables, $x \notin \overline{v_i}$ and $x \notin val(indirect)$. So, like in the base case, variety in x cannot cause variety in y by the execution of the assignment. Moreover, since x is not a member of any of the v_i security variables, we know from the composition property and Γ(S) that variety in x before the start of S cannot lead to variety in the v_is on the right hand side of the := operator. Thus Γ(S) $\Rightarrow$ Γ(S;$y := E(v_i)$). That $x \notin val(indirect)$, we know that x does not influence any paths in the programme so varying x cannot lead to a trace $\mathcal{T}$' which has a different control path and thus a different value for y (see conditional command proofs).

C $=$ [i=1..N $\Box C_i \rightarrow S_i;$]

We assume that y can possibly receive an assignment in one of the branches, otherwise no change in y can possibly occur and Γ(C) trivially holds.

Base case Consider that ([i = 1..N $\Box C_i \rightarrow S_i;$], σ) $\rightarrow^*$ (ϵ, τ). If $x \notin \tau(\overline{y})$, then $\mathcal{M}_S$ guarantees that x is not one of the Boolean guard variables. Thus, $\models C_i(\sigma) = \models C_i(\sigma')$ and $\models \neg C_i(\sigma) = \models \neg C_i(\sigma')$ so the same branch can execute in σ and σ'. So no matter the value of x in σ', there is a trace $\mathcal{T}$' with the same control as $\mathcal{T}$. Therefore the proof obligation is rewritten as,

$$\Gamma([\text{ i = 1..N } \Box C_i \rightarrow S_i;]) \text{ iff } \Gamma(S_i) \text{ i = 1..N}$$

which is just a re-statement of the original proof obligation.

Inductive step This is an extension of the base step; since Γ(S), $x \notin \tau(\overline{y})$ means that x does not belong to the security variable of any variable in the Boolean guards on entry to the command, nor has it affected any programme paths.

$\mathbf{C} = *[\ \text{i=1..N}\ \Box C_i \rightarrow S_i;]$

again we suppose that y can be updated in some branch. Otherwise the proof is trivial.

Base step If $x \notin \tau(\overline{y})$ on termination then x is not in any of the Boolean guards. Thus, $\models C_i(\sigma) = \models C_i(\sigma')$ and $\models \neg C_i(\sigma) = \models \neg C_i(\sigma')$ on each iteration so any trace $\mathcal{T}$' can have the same sequence of iterations. Thus we only need to analyse the branches. Suppose that all guards are initially false; the main part of the repetitive command behaves as the **skip** which has already been proved. If the command executes a branch, then, like for the alternative,

$$\Gamma(*[\ \text{i} = 1..\text{N}\ \Box C_i \rightarrow S_i;]) \text{ iff } \Gamma(S_i)\ \text{i} = 1..\text{N}$$

With the possibility of more iterations we must prove that $\Gamma((S_i);S_j)$ for all j. But to prove this we must first prove $\Gamma(S_i)$. But there being no new proof obligation:

$$\Gamma(*[\ \text{i} = 1..\text{N}\ \Box C_i \rightarrow S_i;]) \text{ iff } \Gamma(S_i)\ \text{i} = 1..\text{N}$$

Inductive step This is an extension of the base step; since Γ(S), $x \notin \tau(\overline{y})$ means that x does not belong to the security variable of any variable in the Boolean guards on entry to the command, nor can a change lead to a different execution path.

Parallelism We suppose that the property holds for some $\vec{S}$, that is

$$\Gamma(S_1 \parallel ... \parallel S_i \parallel ... \parallel S_N)$$

The communication transition is the most interesting.

$$P_j\ !\ exp\ //\ P_i\ ?\ y$$

If $x \notin \tau(\overline{y})$, then we know:

i) $x \notin \{e \cup \overline{e} \mid e \in exp\ \}$
ii) $x \notin val(indirect_i)$
iii) $x \notin val(indirect_j)$

Clause (i) implies that x is not one of the variables v_i in exp, and that like for the assignment, $\Gamma(\vec{S})$ tells us that x has not influenced the variables v_i. The other two clauses tell us that no conditional command skipped a communications due to the value of x - thus changing x in σ' cannot cause another interprocess flow of control and that no repetitive command was influenced by x.

Conditional commands Suppose that y is a variable in one of the branches. If a branch with a Boolean guard executes then x not being in $\overline{y}$ at the end means that x does not influence evaluation of the guards. Again, we need only show $\Gamma(S_i)$. If a branch with a communications guard executes then even if x is a member of the Boolean guards, it need not be registered since the values of the guard do not influence the communication being taken. Thus for all σ' there is a trace $\mathcal{T}$' which can take the branch as does trace $\mathcal{T}$.

Hence, we consider the $\mathcal{M}_S$ flow semantics sufficient to capture the flows that arise.
QED

A.2 Necessity

Again, the proof is by induction. We suppose that the property, denoted Φ holds for an arbitrary programme segment $\vec{S}$', and proceed to show that the property holds when any command is added to the end, $\Phi(\vec{S})$. We treat the sequential case first.

C = Skip

Base step Given that all security variables are initially empty, $\mathcal{M}_S$ cannot possibly produce a state where $x \in \overline{y}$. There is nothing to show necessary.

Inductive step $x \in \overline{y}$ after termination of **skip** if it holds beforehand. Since **skip** produces neither a flow security nor a functional state change $\Phi(\vec{S})$ if $\Phi(\vec{S'})$; but this is guaranteed from the induction hypothesis.

C = $y := E(v_i)$i=1..N

Base step If $x \in \overline{y}$ on termination then x must be one of the right hand side variables; obviously variations in a right hand side variable can cause y, the left hand side variable, to vary.

Inductive step if at the end of the command $y := exp(v_i)$i=1..N, $x \in \tau(\overline{y})$ then either (i) $x \in \{v_i\}$i=1..N, (ii) $x \in \{\overline{v_i}\}$i=1..N or (iii) $x \in val(indirect)$ or any combination of these. If (i) is true then the registration in $\overline{y}$ is necessary since variety can easily be transmitted; if (i) is false but (ii) is true then $\mathcal{M}_S$ is necessary by $\Phi(\vec{S'})$ and the composition property. Finally, the proof of necessity for the case where only (iii) is true is given below for the conditional commands.

C $= [$ **i = 1..N** $\Box C_i \rightarrow S_i;]$

If $x \in \tau(\overline{y})$ where x is not one of the guard variables means that the flow arose in the branch S_i. We must thus show $\Phi(S_i)$ but this is just our proof obligation. **update1** first adds the guard $\overline{C_{bool}}$ to *indirect*. This is necessary because another value for one of these variables may cause the guard to be false which would mean that the branch would not execute and therefore we could not have the same control path. The update to *lhs_vars* comes since changing x may force some other branch to be selected and assign y some other value.

Remark The interesting point is that we cannot justify the necessity of registering flows from the variables in the other guards using our formal definition. Yet the example on page 9 clearly shows that an observer can deduce something. There seems to be a more deductive form of information flow that is occurring here. Hence, we keep the semantics despite of our inability to formally justify it.

C $= *[$ i=1..N $\Box C_i \rightarrow S_i;]$

This is similar to the alternative except that we have **update3**. Here the change to *indirect* is needed since altering an x in any Boolean guard may cause the loop to never terminate - this alters the control path. The changes to *lhs_vars* are needed since for the case where all guards are false - a different x could cause one of these branches to execute and hence alter y, leaving $\tau(y) \neq \tau'(y)$.

Parallelism

Take the communication command first. For $P_j \,!\, exp \,//\, P_i \,?\, y, x \in \tau(\overline{y})$ at the end implies that $x \in \{e \cup \overline{e} \mid e \in vars(exp)\}$ and/or x is either of the processes indirect variables. The proof of $\Phi(\vec{S})$ if the first condition is met, is the same as for assignment; if only the second condition holds, then the proof of $\Phi(\vec{S})$ comes from the conditional commands. For the special communication case where the process index is registered in the remote *indirect* - this is

justified since changing i can alter the control flow of the programme making it impossible to have a $\mathcal{T}$' with the same control path as $\mathcal{T}$.

Conditional Commands For the alternative, **update1** adds the Boolean guard variables if the executing branch is Boolean guarded. Supposing x is in the guard, a new value for x might invalidate the truth of the guard and thus force some other branch - even one with a communications guard to execute. Thus registering x in all branches is necessary. For **update2**' x is added to *indirect* if another branch has a communication. This is important since a different x forcing another branch to execute could take that communication which would alter the control path through the programme.

QED

B Proof that the proof system respects the flow semantics

Aside Even though the security proof system is based on [Apt et al.80], the proofs of soundness and completeness are much simpler than Apt's (see [Apt 83]). The reason for this is that the security proof system does not contain the notion of a bracketed section or invariant. More precisely, in our proof system, a bracketed section contains a communications command only, hence each sub-programme is *normal* and thus each parallel composition *admissible*. Given these conditions, even the proofs in [Apt 83] become easy. $\diamond$

B.1 Soundness

By soundness we mean that if a programme S produces a flow security state satisfying a predicate p, then the proof system must not show that $\neg p$ is true. More precisely,

For the axioms if $\vdash$ {p} S {q} then, $\forall\ \sigma, \tau$ such that $\models p(\sigma)$ and $\tau = \mathcal{M}_S$[S](σ), then $\models q(\tau)$

For the rules if a_0, ..., $a_n \vdash$ {p} S {q} where the a_is are antecedents, $\forall\ \sigma, \tau$ such that $\models p(\sigma)$ and $\tau = \mathcal{M}_S$[S](σ), then $\models q(\tau)$, assuming each a_i to be true.

We firstly give the substitution lemma which is crucial in the proofs.

Substitution lemma For all predicates P on the flow security state, and for all states σ; let $\overline{x}$ be a security variable or the variable *indirect*, and E be a set of variables,

$$\mathrm{P}[\overline{x} \leftarrow \mathrm{E}](\sigma) = \mathrm{P}(\sigma{+}(\overline{x} \rightarrow \mathrm{E}))$$

where $\mathrm{P}[\overline{x} \leftarrow \mathrm{E}]$ is a predicate similar to P except that every free occurrence of the variable $\overline{x}$ is replaced by the set E. We omit the proof.

Skip This is trivial to prove. Since $\sigma{=}\mathcal{M}_S[\mathbf{skip}](\sigma)$, if $\models p(\sigma)$ then $\models q(\sigma)$ where $p = q$.

Assignment

$$\sigma\text{'} = \mathcal{M}_S[y := Exp(x_1, x_2, ..., x_N)](\sigma)$$

where $\sigma\text{'} = \sigma{+}(\overline{y} \rightarrow (\{x_i\} \cup \sigma.\overline{x_i} \cup val(\sigma.indirect)))$ for i=1..N. Let us denote the new value of $\overline{y}$ as E. For our axiom $A_{:=}$, if the pre-condition holds in state σ, then $\models p[\overline{y} \leftarrow E](\sigma)$. For post-condition p to be met, $\models p(\sigma')$ must be true. But, $\models p(\sigma')$ is equivalent to $\models p(\sigma + (\overline{y} \rightarrow E))$ from $\mathcal{M}_S$, which must be true since it is equivalent to $\models p[\overline{y} \leftarrow E](\sigma)$ according to the substitution lemma.

Composition The flow semantics have the following property: $\mathcal{M}_S$[S1;S2](σ) is equivalent to $\mathcal{M}_S$[S2]($\mathcal{M}_S$[S1](σ)). We let $\sigma' = \mathcal{M}_S$[S1](σ). Then $\mathcal{M}_S$[S1;S2](σ) $\equiv \mathcal{M}_S$[S2]($\mathcal{M}_S$[S1](σ)) $\equiv \mathcal{M}_S$[S2](σ') $\equiv \tau$ We know from our first antecedent that $\models q(\sigma')$ since $\models p(\sigma)$. Our second antecedent tells us that since $\models q(\sigma')$ and given $\mathcal{M}_S$[S2], then $\models r(\tau)$. Thus, for $\tau = \mathcal{M}_S$[S1;S2](σ) and $\models p(\sigma)$ we deduce $\models r(\tau)$ and so the composition rule is sound.

Alternative The soundness of the composition rule allows us to deduce the soundness of {p} [i=1..N $\Box C_i \rightarrow S_i$;] {q} given that the antecedents are {p} **update1** {r}, {r} S_i {t} and {t} **update2** {q} and $\mathcal{M}_S$[i=1..N $\Box C_i \rightarrow S_i$;] $= \vee_{i=1}^{N}$ $\mathcal{M}_S$[**update1**;S_i;**update2**].

Repetitive We let σ_k'=$\mathcal{M}_S$[rep^k](σ) for $k \geq 0$. (where rep^k denotes k repetitions of **update1**;S_i;**update2**). When k is zero, then there are no iterations and the only alterations are to *indirect* and the *lhs_vars*, i.e. **update3**. Thus $\mathcal{M}_S$[rep^0](σ) = σ+(**update3**). When $\models p(\sigma)$ then the {p} **update3** {q} antecedent tells us that $\models q(\tau)$. For $k > 0$, we rely on the fact that the antecedents tell us that the predicate p is established on each iteration. The remaining proof reasoning is the same as for the alternative command.

Parallelism Again, what we want to prove is that for $\vdash$ {P} $\vec{S}$ {Q}, then if $\tau \in \mathcal{M}_S[\vec{S}](\sigma)$ and $\models$P(σ) then we have $\models$Q(τ). Since the free variables in P and Q are *disjoint*, that is the free variables for one process are not the same for any other, the formula $\models$P(σ) can be rewritten as $\models \wedge_{i=1}^{N} p_i\ (\sigma)$ and $\vdash$ {P} $\vec{S}$ {Q} rewritten as $\vdash \{\wedge_{i=1}^{N} p_i\}\ \vec{S}\ \{\wedge_{i=1}^{N} q_i\}$. Thus it suffices to show soundness for each process in turn. We only need consider the communications command here.

Firstly, we re-write the communications axiom to express it more like the assignment axiom [Hoare69].

$$\{p_1 \wedge p_2[\overline{y} \leftarrow \{x\} \cup \overline{x} \cup val(ind_1) \cup val(ind_2),\ ind_1 \leftarrow ind_1 \sqcup ind_2,\ ind_2 \leftarrow ind_2 \sqcup ind_1]\}$$
$$P_1!x\ //\ P_2?y$$
$$\{p_1 \wedge p_2\}$$

The soundness proof for the communication axiom is similar to the assignment proof (and is based on the substitution lemma) so we omit it.

The parallel versions of the alternative and repetitive commands follow easily. For example, for the alternative command, the proof reasoning for the sequential composition rule allows us to say that for some branch with a communications guard: {P} $C_i \rightarrow S_i$ {Q} is sound if {P} C_i {T_i} and {T_i} S_i {Q} are correct. The argument for the repetition command is the same.

This concludes the proof of soundness.
QED

B.2 Completeness

Aside The security semantics say nothing about the functional properties of the programme. In this sense they are "weak"; for example, for the alternative command we say that if some branch S_i terminates, then **update2** is made to the flow security state. This allows us to talk about the completeness of the proof system rather than the relative completeness. $\diamond$

A proof of completeness verifies that if some formula $\models$ {p} S {q} is true, then the proof system can show it to be true, that is $\vdash$ {p} S {q}. We repeat the consequence rule as it is crucial for the proof.

$$\frac{\text{P}\Rightarrow\text{P'},\ \{\text{P'}\}\ \text{S}\ \{\text{Q'}\},\ \text{Q'}\Rightarrow\text{Q}}{\{\text{P}\}\ \text{S}\ \{\text{Q}\}}$$

skip We have $\models$ pre(**skip**, p) **skip** {p}, where pre(**skip**, p) denotes the set of states which can exist before the execution of the **skip** command if the predicate p is to hold on termination.

$$\begin{aligned}\text{pre}(\textbf{skip}, \text{p}) &\hat{=} \{\sigma \mid \forall\tau, \models p(\tau) \wedge \tau{=}\mathcal{M}_S[\textbf{skip}](\sigma)\} \\ &= \{\sigma \mid \forall\tau, \models p(\tau) \wedge \tau{=}\sigma\} \\ &= \{\sigma \mid \models p(\sigma)\} \\ &= \text{r, r} \Rightarrow \text{p}\end{aligned}$$

then $\models$ {r} **skip** {p} for r$\Rightarrow$p only, hence $\vdash$ {p} **skip** {p} is justified since we also have the consequence rule.

Assignment Suppose that $\models$ {p} y:=Exp$(x_1, x_2, ..., x_N)$ {q}. Then, $\exists\ \sigma$, where $\models p(\sigma)$ such that $\tau{=}\mathcal{M}_S$[y:=Exp$(x_1, x_2, ..., x_N)$](σ) and $\models q(\tau)$. Given $\mathcal{M}$[y:=Exp$(x_i)i = 1..N$] we know that $\models q(\sigma + (\overline{y} \rightarrow E)$. The substitution lemma tells us that $\models q[\overline{y} \leftarrow E](\sigma)$ where E is the flow value. Thus our assignment axiom is complete since we have $\vdash$ {p} y:=Exp(x_i)i=1..N {q} where p logically implies $q[\overline{y} \leftarrow E]$ (using the consequence rule as well).

Composition Suppose that we have $\models$ {p} S1;S2 {r} from $\mathcal{M}_S$. We want to show that we can establish $\vdash$ {p} S1;S2 {r} with the proof system. Firstly we give the following assertions: $\models$ {p} S1 {post(p,S1)} where post(p,S1) is the set of all states that can be established after S1 if p holds beforehand, and $\models$ {pre(S2,r)} S2 {r} where pre(S2,r) is the set of states which can exist beforehand if r is to hold afterwards. Since $\models$ {p} S1;S2 {r}, there must be a set of states common to post(p,S1) and pre(S2,r). We denote these states q:

$$\begin{aligned}\text{q} = \text{post(p,S1)} \cap \text{pre(S2,r)} \\ \{\ \sigma' \mid \sigma' = \mathcal{M}_S[\text{S1}](\sigma) \wedge \tau{=}\ \mathcal{M}_S[\text{S2}](\sigma')\ \}\end{aligned}$$

Hence, $\models$ {p} S1;S2 {r} can be rewritten as $\models$ {p} S1 {q} and $\models$ {q} S2 {r} so that $\vdash$ {p} S1;S2 {r} can be shown if $\vdash$ {p} S1 {q} and $\vdash$ {q} S2 {r} which is equivalent to our composition rule.

Alternative As shown in $\mathcal{M}_S$, this is equivalent to the triple **update1**;S_i;**update2**. Thus $\models$ {p} [i=1..N $\Box C_i \rightarrow S_i$;] {q} implies that there are some predicates r and t such that $\models$ {p} **update1** {r}, $\models$ {r} S_i {t} and $\models$ {t} **update2** {q}. That $\vdash$ {p} **update1** {r} and $\vdash$ {t} **update2** {q} from $\models$ {p} **update1** {r} and $\models$ {t} **update2** {q} is easily shown; the proof is very similar to the assignment proof. Thus the alternative command is complete if we can show $\vdash$ {r} S_i {t} from $\models$ {r} S_i {t}. This is just a reduced version of our proof obligation.

Repetitive Suppose that $\models$ {p} *[i=1..N $\Box C_i \rightarrow S_i$;] {q}. Then $\exists$ p' such that $\models$ {p} rep^k {p'} and $\models$ {p'} **update3** {q} (where rep^k denotes k repetitions of **update1**;S_i;**update2**). For $\models$ {p} rep^k {p'}, 0 iterations implies $\models$ {p} rep^0 {p} since it is equivalent to a **skip** and we do have $\vdash$ {p} rep {p}. For $k > 0$; we write

$$\models \{p\}\ rep^k\ \{post(p,rep)\}$$

where post(p,rep) is the set of all states that can exist after k branches execute. But because our semantics are "weak", a formula is valid only if it holds no matter how many iterations, then,

$\models$ {p} rep^k {post(p,rep)}
$\equiv$ {p} rep^{k-1}; {post(p,rep)} **update1**;S_i;**update2** {post(p,rep)}
$\Rightarrow$ p={post(p,rep)}.

Thus we need an invariant on the repetitive part **update1**;S_i;**update2**. In $R_{s-repetitive}$, p serves as this invariant. That we can get $\vdash$ {p'} **update3** {q} from $\models$ {p'} **update3** {q} follows from the assignment proof.

Parallelism We want to show that for each correct formula $\models$ {p} $\vec{S}$ {q}, we can get $\vdash$ {p} $\vec{S}$ {q}. The first step is to re-write the formula as $\models$ {p_i} S_i {q_i}, this being permitted by the disjointness property on the processes' predicates. Thus we must show that we can get $\vdash$ {p_i} S_i {q_i}.

The resulting proof obligation is similar to the sequential case. Only the communication axiom remains to be treated. Let α be a communication command, since $\mathcal{M}_S[\alpha]$ is undefined (that is, it only has a meaning when paired with another communication command), $\models$ {p_1} α {q_1} if and only if there exists a paired communication β for which $\models$ {p_2} β {q_2} such that $\models$ {$p_1 \wedge p_2$} $\alpha//\beta$ {$q_1 \wedge q_2$}. The proof that $\vdash$ {$p_1 \wedge p_2$} $\alpha//\beta$ {$q_1 \wedge q_2$} using the communication axiom is very similar to the assignment proof.

QED

Contents

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399