An Approach to Information Security in Distributed Systems

C. Bryce, J-P. Banâtre, D. Le Métayer
IRISA/INRIA-Rennes,
Campus de Beaulieu,
35042 Rennes Cedex,
France.

Abstract

Information flow control mechanisms detect and prevent illegal transfers of information within a computer system. In this paper, we give an overview of a programming language based approach to information flow control in a system of communicating processes. The language chosen to present the approach is CSP. We give the security semantics of CSP and show, with the aid of examples, how the semantics can be used to conduct both manual and automated security proofs of application programs.

1 Introduction

One of the most noteworthy trends in modern computing is the proliferation of automated systems to commercial, administrative and other fields. The amount of sensitive information being stored in computer systems has increased as a consequence and with that, the seriousness of an *attack* on a system where a user succeeds in illegally gaining access to information in the system. Distributed computing has excacerbated this problem since information can be accessed over wide geographical distances and so not only is there more means of attack, but it is more difficult to control the processing activity of a system which leads to these attacks on the security, or *confidentiality*, of the information stored.

Computer systems typically ensure information confidentiality using *access control* mechanisms [9]. In Unix for example, each file has an access control list specifying the access rights (*rwx*) that the owner, the owner's group and other users (denoted 'world') possess for the file. The basic principle of access control models is to associate an access key with each operation on an object; only a process possessing a key may execute the associated operation on the object. Yet as the example in figure 1 illustrates, information confidentiality cannot be assured by access controls alone. In this example, user JOHN creates a mail message FYEO, denoting "for your eyes only",

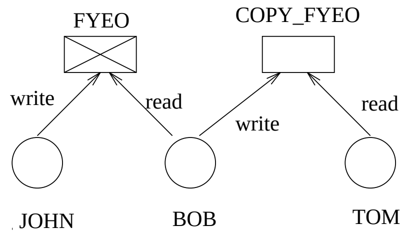

Figure 1: Insecure Information Flow

which he wants to send to user BOB. To do this, JOHN creates the object FYEO and gives a key for the *read* operation to BOB. To ensure that user TOM does not read the mail message, JOHN can specify that TOM must not receive the *read* right for FYEO. Nevertheless, BOB may copy the contents of FYEO and write them to file COPY_FYEO, a file for which TOM possesses a *read* right. The result is that an illegal *information flow* from FYEO to TOM occurs which the access controls have not prevented. To prevent this type of information flow, one could structure the access constraints of the system in such a way that BOB can never write to a file that can be read by TOM; however, this means that no sharing of information can ever occur between these two users. Rather, a mechanism is needed which allows the flow of information in the system to be controlled.

In the preceding example, there is an implicit assumption that an attacker, in this case TOM, has an understanding of how the system behaves, that is, he expects that the data written to the file COPY_FYEO are the contents of the FYEO message sent to BOB by JOHN. Note that BOB may not necessarily have wilfully divulged the contents of this message: his mail software may be errorness or infected by corrupt software known as a *Trojan Horse* [8]. That such an understanding of the system's behavior can be known by users is very plausible, given that common user programs, such as editors, mail programs and file management utilities are often public domain software packages, available from any number of FTP sites. The information security problem that must be tackled is generalized in the schema of figure 2.

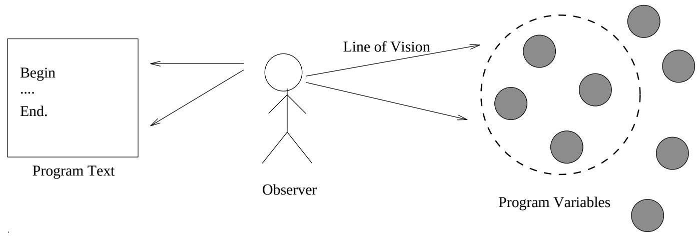

Figure 2: Observer inferring Information

An observer possesses a copy of the code of the application (e.g., mail program) - this is represented in the left hand side of the figure. The observer also sees what is happening at run-time, that is, the security policy of the system allows him to read the values of certain variables (e.g., mail objects). In the figure, the variables which the observer is permitted to see are marked as shaded disks within the dotted circle; the disks outside of this circle represent the variables of the application program, the information in which the observer is forbidden to know. The security requirement is the following: given that the observer knows the program text of the system, he must not be able to infer the value of the variables outside of the circle from the value of those variables inside; in other words, there must be no information flow from the variables outside the circle to those on the inside.

As an example of the problem being addressed, let x be a variable outside of the circle - a variable whose value the security policy of the system forbids the observer to know; let y be a variable inside the circle. If a program containing the single assignment $y := x$ is executed, then an illegal information flow occurs to the domain of the observer: the observer can deduce the value of x from the value of y.

This paper's approach to tackling this problem is to take a standard parallel programming language - we choose CSP [11] - and to define the set of information flows that each command of the language effects. By formalizing the description of these flows, the *information flow semantics* of the language is defined. This is the subject of section 2. Section 2 also details the semantics in axiomatic form, that is, as a series of axioms and rules [10] which are used to manually prove that information flow restrictions placed on a program hold. In section 3, the information flow semantics are developed into a compile-time algorithm that can automatically verify the information flow constraints of a (sequential) program. Section 4 looks at related work and gives some conclusions.

2 An Information Flow Security Proof System

CSP [11] is a parallel programming language developed by Tony Hoare; this language is analyzed for information flow in the following discussion. The main language declarations are given in figure 3.

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*

Figure 3: CSP Syntax Declarations

In the figure, $\prec\succ$ 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 program 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 of 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 with it. 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.

2.1 Direct & Indirect Information Flows

There are two classes of information flows in programs. 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 [6]. More intuitively, an observer learns more about the right-hand side variables after execution of the command. The information that was in the destination variable is lost.

Conditional commands introduce a second class of flows [7]. The fact that a command is conditionally executed transfers information to an observer on the value of the command guard. Consider the following program segment. We use a multiple assignment for brevity; $e()$ is some expression:

```
$x := e();$
$a := 0;\ b := 0;$
[ $x = 0 \rightarrow a := 1$
  $\square\ x \neq 0 \rightarrow b := 1$ ]
```

If someone knows the program text then, by inspecting either a or b after program execution, an observer can deduce whether x was zero or not. This is an example of an implicit flow [7] or what we will more generally refer to as an *indirect* flow.

To reason about the information flows of a program, some way of representing these flows is needed. We notably 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, of which there is one per process, and is 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 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:

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

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.

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

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:

$$indirect_i \sqcup indirect_j \;\mathrel{\hat{=}}\; (val(indirect_i) \cup val(indirect_j)) \uplus indirect_i$$

The *flow security state* of a program is defined as firstly, the mapping from each variable to its security variable and secondly, the value of *indirect*. Note 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 behavior of the command types with respect to the flow security state, we will use an operational notation similar to [13]. The flow semantics are defined as a transition relation "$\longrightarrow$" which maps a program segment and state pair to another such pair. The interpretation given $(C_1, \sigma) \longrightarrow (C_2, \tau)$ is that the execution of the command structure C_1 in flow security state σ leads to a state τ from which the command structure C_2 executes.

2.2 Assignment Command Flow Semantics

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

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 a member of the security variable $\overline{y}$ since y's new information content is independent of (any previous) content of x. An alternative approach, where instances of variables in the program are tagged is described in [5]; this mechanism permits the instances of a variable at various points of the program to be differentiated.

The effect of the assignment on the program security state is captured by the following axiom (à la Hoare [10])[2]:
$A_{:=_s}$

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

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. This axiom generalizes assignments with array variables since the commands $a[i] := e$ and $e := a[i]$ are equivalent to $a := exp(a, i, e)$ and $e := exp(a, i)$ respectively.

2.3 Sequential Composition of Commands

The rule for sequential composition (;) follows. If the command S1 establishes a flow security 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:
$R_{s-composition}$

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

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

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

2.4 Alternative Command Flow Semantics - Sequential Case

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, command guards can be related, so a guard being true may imply the value of other guards. Thus, we consider that there is an indirect flow from the variables of 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 program segment.

```
$x, y := 0, 0;$
[ $B \rightarrow x := 7; y := 9$ □ **not** $B \rightarrow y := 2$ ];
[ $x = 7 \rightarrow S_1$ □ $y = 9 \rightarrow S_2$ □ $x = 0 \rightarrow S_3$ ];
```

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, an observer knows 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 behavior of a secure alternative command with respect to the flow security state can be described as follows. For a given flow security state σ,

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

where

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

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. Naturally, the branch executed depends on the functional state.

The transition is explained as follows. 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 $\overline{C_{bool}} = \{c \cup \overline{c} \mid c \in C_{bool}\}$;
$R_{s-alternative}$

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

The rule is geared towards flow security proofs. Hence, for a predicate R, there is a predicate T that is established no matter which of the command branches S$_i$ is executed. The relation between P and R and between T and Q is that engendered by the assignment semantics and captures those modifications made to the flow security state in `update1` and `update2`.

2.5 Repetitive Command Flow Semantics

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 an observer of these variables 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 */
$z, y, t := 0, 0, 0;$
*[ $x \neq y \rightarrow y := y + 1$
 □ $r \neq t \rightarrow t := t + 1$ ];
$z := 1;$
```

In this program 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 Reitman [14] points 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 behavior of the repetitive command with respect to the flow security state:

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

where

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

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:
$R_{s-repetitive}$

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

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. After termination, the final modifications to the flow variables establish a state satisfying Q.

2.6 Communication Command Flow Semantics

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 *indirect*s 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, that is, an observer of each process is aware that the other process has communicated.

$$(P_2!x \parallel P_1?y,\ \sigma) \rightarrow (\epsilon,\ \sigma')$$

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 $(indirect_2 \sqcup indirect_1)$ of σ.

2.7 Parallelism

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 these communications take place, and that the updates which follow are made, gives information to observers of other processes about the condition that was met in the process that made the first rendezvous.

Take the example in figure 4. An observer sees what is happening in process P2 at run-time. Since he knows the process text, he can deduce the following results: If the value of y changes, then a must be zero in process P1; if $b \neq 0$ and r becomes 1, P1 did not take the branch with the communication command in its alternative command and so $(a \neq 0)$ must hold. Thus, indirect inter-process information flows may occur without a communication taking place.

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.

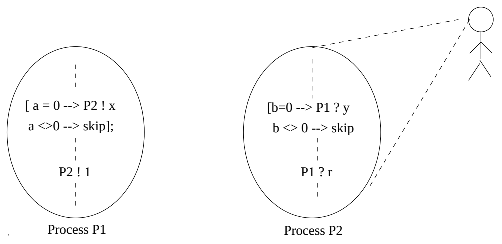

Figure 4: Parallelism and Information Flow

2.7.1 The Alternative command

The complete alternative command of CSP is more complex than that introduced earlier. 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 $b; \alpha \longrightarrow S$ to be rewritten as $b \longrightarrow \alpha; S$ where b is a non-constant Boolean expression, 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 summarized:

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.

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

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.

```
update2' ≙ if $missed\_comms$
           then $indirect := head(indirect) \uplus tail(indirect)$
           else $indirect := tail(indirect)$
           endif
```

And for a branch with a communications guard.

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

if there exists a matching communication command α, such that $(\alpha \parallel C_i,\ \sigma) \longrightarrow (\epsilon,\ \sigma\text{'})$.

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 rendezvous mechanism (**update2**', then part).

We derive the following rule:
$R_{s-alternative}$

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

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

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 program 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 analyzing a program to know which guards are dependent; the predicates may not be always be able to give us that information.

2.7.2 The Repetitive command

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

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

and for a communications guard ...

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

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 $R_{s-alternative}$:
$R_{s-repetitive}$

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

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

2.7.3 Proving Systems Information Flow Secure

The axioms and rules given permit us to prove that systems written in the CSP programming language are infomation flow secure. Like for correctness proofs of parallel programs [2], a two-staged approach to proving parallel programs secure is used. In the first stage, each process is proved individually with assumptions being made on the values of the information flows exchanged between processes during communication. The second stage validates these assumptions.

Given that the proof system makes assumptions about the information flow exchanges made by a process in the first stage, in both the send and receive command axioms, any post-condition can be established:

A_{s-send}

$$\{ \text{ P } \} \prec \text{Process} \succ \,!\, \prec \text{Expression} \succ \{ \text{ Q } \}$$

$A_{s-receive}$

$$\{ \text{ P } \} \prec \text{Process} \succ \,?\, \prec \text{Variable} \succ \{ \text{ Q } \}$$

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

$A_{s-communication}$

$$\begin{array}{c} \{\ z_1 = indirect_1,\ z_2 = indirect_2\ \} \\ P_2!x \ \| \ P_1?y \\ \{\ \overline{y} = \{x\} \cup \overline{x} \cup val(indirect_1) \cup val(indirect_2),\ indirect_1 = z_1 \sqcup z_2,\ indirect_2 = z_2 \sqcup z_1\ \} \end{array}$$

The second phase of a CSP program flow security proof demonstrates that the proofs of each process P_1, P_2,....., P_N *co-operate*:

$R_{co-operation}$

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

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 $\succ \{q_j\}$,

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

Note how condition (2) says *semantically* matching communication commands, that is, commands which name each other's process and which can possibly communicate during program execution. It may seem strange that semantically paired commands are stated in the rule since the flow security state does not capture the semantically matching commands. This is another area where a static security analysis is aided by a functional analysis.

Simple Example As an example of how the proof system works, consider a program where process A sends secret information to process C via process B. The code and the predicates that we want to prove are given in figure 5. We must be able to show that process C receives illegal information - that the predicate $secret\_val \in \overline{y}$ is true. Since all commands of each process are simple communication commands, each process is individually proved using the fact that the communication command axioms permit any post-condition, and so the first phase of the proof terminates. Note also that since no process contains a conditional statement, the *indirect*s of each are empty. The communication axiom means that the predicate $\overline{x}$=$\{secret\_val\}$ must hold after the first of the commands in process B; this axiom also enforces that the predicate $\overline{y}$=$\{secret\_val,\ x\}$ holds after the communication command in process C. Since the latter predicate implies the chosen post-condition, the program is proved.

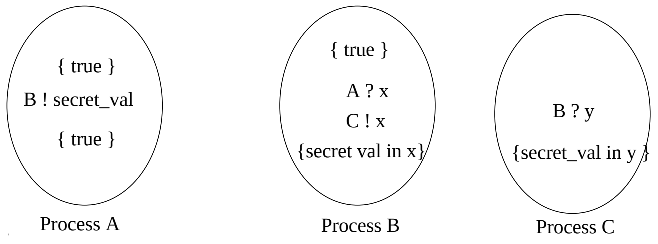

Figure 5: Simple Information Flow Proof Outline

2.8 Example : a Secure Mail System Service

The following example is typical of many distibuted system applications. We consider a mail system with a central server. The server receives messages from users and passes them on to the destination users. Moreover, so that the system administrator has enough information to make decisions pertaining to resource usage, the server maintains a log of who sends messages to whom. To guarantee information confidentiality for the mail service's clients, the system administrator must not be able to read any mail message that passes through the central server.

An outline of the code of this system is given below.

```
client_send[i] ::                      client_receive[i] ::
var dest, smsg;                        var source, cmsg;
*[ console ? dest →                    *[ server ? source →
   console ? smsg;                        server ? cmsg;
   server ! dest;                         console ! cmsg;
   server ! smsg; ]                       console ! source; ]
||                                     ||
server ::
var message, dest, i, req;
var no_comms; array send_comms, dest_comms;
message := 0; no_comms := 0; i := 0; dest := 0; req := 0;
*[ SysAdmin ? req → SysAdmin ! send_comms; SysAdmin ! dest_comms;
□ no_comms < 100 → client_send[i] ? dest;
                   client_send[i] ? message;
                   no_comms := no_comms + 1;
                   send_comms[no_comms] := i;
                   dest_comms[no_comms] := dest;
                   client_receive[dest] ! i;
                   client_receive[dest] ! message; ]
```

On the client side, the mail application contains two processes, client_send and client_receive, for sending and receiving messages from the mail server respectively. The mail server accepts up to 100 messages from clients, registers the sender and receiver and then transmits the message. The server also accepts requests from a SysAdmin process (not shown) for mail service usage information.

Security Requirements & Proof The system is secure if the system administrator cannot know the text of the actual messages that pass through the service. Thus, the data passed to the administrator, send_comms and dest_comms, must not have received an information flow from the message variable of any client_send process:

$$\textbf{Sec\_Req} \mathrel{\hat{=}} \text{client\_send[i].message} \notin \overline{\textit{send\_comms}} \cup \overline{\textit{dest\_comms}} \cup \text{val}(\textit{indirect})$$

is always true in the server, that is, the invariant P of the server loop ensures P. We choose P as this invariant:

$$\text{P} \mathrel{\hat{=}} \{\ \{\text{cs\_dest}\} \subseteq \overline{\textit{dest}},\ \{\text{cs\_msg}\} \subseteq \overline{\textit{message}},\ \{\text{no\_comms}\} \subseteq \overline{\textit{no\_comms}},\ \{\text{send\_comms}\} \subseteq \overline{\textit{send\_comms}},$$
$$\{\text{dest\_comms}\} \subseteq \overline{\textit{dest\_comms}},\ \{\text{dest, no\_comms, dest\_comms, cs\_dest}\} \subseteq \overline{\textit{dest}},\ \{\text{no\_comms}\} \in \textit{indirect}\ \}$$

since **Sec_Req** $\Rightarrow$ P, we just need to show that P is indeed the invariant, and pre-condition, of the server's loop. *cs_dest* denotes the *destination* variable of the client_send process. The initial assignments in the server establish a state satisfying:

$$\text{pre} = \{\ \overline{\textit{all}} = \{\},\ \textit{indirect} = \textit{prec}\ \{\} \succ\ \}$$

where *all* stands for each of the process' variables. The assigment axiom is used to verify this. For example, {pre'} req := 0 {pre}, holds since $A_{s-:=}$ means that the pre-condition holds if and only if pre $\Rightarrow$ pre$[\overline{\textit{req}} \leftarrow \text{val}(\textit{indirect})]$. The value of val(*indirect*) is {}, so the pre-condition pre' given by the axiom is pre with the sub-predicate for $\overline{\textit{req}}$ removed. Following through this reasoning for each of the command, the predicate arrived for the server's pre-condition is val(*indirect*) = {}; this must be true since the *indirect* of each process is initially empty. The result is that P is a satisfactory loop pre-condition; we need now only show that P is also a loop invariant.

There are two branches in the repetitive command, the first one being guarded by a communications command; the command list of this branch only consists of communications commands. Following the repetitive command and the axioms for the communications commands, we can let P be the post-condition of this sequence of commands since any post-condition can be chosen. That P is a correct post-condition will be proved in the second state of the proof.

The second branch of the repetitive command is guarded by a Boolean expression. From the repetitive rule, we define an inner invariant R as:

$$R = \text{P } \textbf{but } \mathit{indirect} = \prec \{\text{no\_comms}\} \; \{\} \succ \}$$

R is the same predicate as P except that the value for $\mathit{indirect}$ is $\prec \{\text{no\_comms}\} \; \{\} \succ$. R directly from the repetitive rule, using the fact that $\overline{C\_bool} = \{\text{no\_comms}\}$. The last two commands of the branch list are communication commands and we choose R as their pre-condition, assuming that R is the branch post-condition, which would be true if R were the inner invariant. The third last command of the branch is the assignment dest_comms[no_comms] := dest. R is the post-condition, we let R' be the pre-condition. From $A_{s\_:=}$, R' is defined as:

$R[\overline{dest\_comms} \leftarrow \{\text{dest\_comms, no\_comms, dest}\} \cup \overline{dest\_comms} \cup \overline{dest} \cup \overline{no\_comms} \cup \text{val}(\mathit{indirect})]$
$R[\overline{dest\_comms} \leftarrow \{\text{dest\_comms, no\_comms, dest, c\_msg}\}]$
R **but** dest_comms **removed**

The predicate P **but** v **removed** is the predicate P but without any sub-predicate on the variable v. Following this approach for the two other commands, the pre-condition to the no_comms := no_comms + 1 command, we get

$$R \textbf{ but } \text{dest\_comms, send\_comms } \textbf{removed}$$

Since $R \Rightarrow R$ **but** dest_comms, send_comms **removed**, we insert R as the pre-condition using the standard consequence rule. Finally since the first two commands of the branch command list are communication commands, we claim that R is a suitable post-condition.

Thus R is preserved by both branches - it is an inner invariant. Since R was derived from P using $R_{s-repetitive}$, P is indeed the loop invariant and so the server process is proved so long as all assumptions made about the communication commands are true. That these assumptions are true can be proved in the second stage of the proof which now follows.

The client processes have only communication commands in their loop bodies. In the sending client processes, we choose val($\mathit{indirect}$) $\subseteq$ {no_comms} as the invariant of the loop; in the receiving client process, we choose as invariant:

$$\text{cs\_msg} \subseteq \{\text{message, cs\_msg, dest, send\_comms, dest\_comms}\}$$

We now outline that the proofs of the processes co-operate. Take the communication (*client_send*[i]) || (*server* ! s_msg) as an example. The communication axiom requires that the following post-conditions hold:

$\overline{message} = \{\text{c\_msg}\} \cup \text{val}(\mathit{indirect}_{server}) \cup \text{val}(\mathit{indirect}_{client-send}) \subseteq \{\text{no\_comms, c\_msg}\}$
$\mathit{indirect}_{server} \subseteq \{\text{no\_comms, c\_msg}\}$
$\mathit{indirect}_{client-send} \subseteq \{\text{no\_comms}\}$

which is verified in the post-conditions that we have chosen. The proof of the other communications is similar. In showing that the process proofs co-operate, we can be satisfied that the assumptions made in the first step in the proof of the server process are valid, and thus that the loop invariant is true. Since this invariant satisfies the security requirements, the mail system is proved secure.

2.9 Meaning & Flexibility of Semantics

We have presented an axiomatic proof system for verifying confidentiality properties of parallel programmed systems. This proof system is based on a security semantics, though up until now, we have not been precise about the foundations of this semantics. A concept such as "can infer information" is not precise enough to permit rigorous proofs since there is no way to gage whether the semantics are correct. The precise definition of information flow which we have in fact been using is the following.

Suppose a program Prog with the following functional specification:

$$\{ \; x = x_0 \; \} \text{ Prog } \{ \; y = Y_0 \; \}$$

that is, when the initial value of the variable x is x_0, then the final value of variable y is one of the values in the set Y_0. (Since a CSP program is non-deterministic, a variable may have more than one possible final value). Suppose now that an information flow analysis confirms the following predicate:

$$\{ \text{ true } \} \text{ Prog } \{ \; x \notin \overline{y} \; \}$$

In [5] it is proven that the following predicate on the functional state must then hold:

$$\{ \ x \neq x_0 \ \} \text{ Prog } \{ \ \mathrm{y} = Y_0 \ \}$$

that is, changing the initial value of x can in no way influence the final value of y, nor prevent the program from terminating. The proof is based on the functional semantics of CSP [11]. In contrast, when the security proof system implies that $x \in \overline{y}$, then the final value of y need not be in the set Y_0; indeed, the program may even be prevented from terminating.

Another feature of our proof system is that it registers information flows to a variable in terms of the set of information flow source variables. One can imagine that in the proof of a large system, the size of the security variables, and consequently of the program predicates, becomes large. Moreover, security policies are often expressed in terms of processes or process groups that may exchange information. In military secure systems for instance, processes are classified as high-level or low-level. The security requirement of these systems is that low-level users may not deduce high-level information, in other words, variables of low-level processes may not receive information from variables of high-level processes. To facilitate security proofs of systems with such policies, [5] presents a set of re-writing rules for the proof system; in its re-written version, the semantics registers flows in terms of the processes that send flows. For example, the assigment $x := y$ has as security semantics: $\overline{x} := \mathit{self} \cup \overline{y} \cup \mathit{val(indirect)}$ where *self* denotes the name of the containing process.

3 Detection of Security Leaks at Compilation-Time

An algorithm is presented in [4] whose aim is to verify the information flow security of sequential (CSP) programs at compilation-time. We give a summary of this work here. The principal goal in developing compile-time analysis techniques is to find a deterministic, and hence mechanisable, algorithm. The proof system which we have presented is an unsuitable starting point since it contains two sources of non-determinism: i) the relation between the P and Q predicates of the assignment command's axiom is not one-to-one, and ii) the consequence rule can be applied at any point in a proof.

$$\{R\} \ y := exp(x_1, x_2, \ldots.., x_n) \ \{T_y(R)\}$$

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

$$\frac{\forall i = 1..n\{P\}S_i\{Q_i\}}{\{P\}[C_1 \rightarrow S_1 \square C_2 \rightarrow S_2 \square \ldots\ldots \square C_n \rightarrow S_n]\{\bigsqcup_i Q_i\}}$$

$$\begin{array}{c}
\forall i = 1..n\{P^0\}S_i\{Q_i^0\}, \quad Q^0 = \bigsqcup_i Q_i^0, \quad Q^0 \not\Rightarrow P^0, \quad P^1 = P^0 \bigsqcup Q^0 \\
\hline
\forall i = 1..n\{P^1\}S_i\{Q_i^1\}, \quad Q^1 = \bigsqcup_i Q_i^1, \quad Q^1 \not\Rightarrow P^1, \quad P^2 = P^1 \bigsqcup Q^1 \\
\hline
\vdots \\
\forall i = 1..n\{P^{n-1}\}S_i\{Q_i^{n-1}\}, \quad Q^{n-1} = \bigsqcup_i Q_i^{n-1}, \quad Q^{n-1} \Rightarrow P^{n-1}, \quad P^n = Q^{n-1} \\
\hline
\{P^0\} * [C_1 \rightarrow S_1 \square C_2 \rightarrow S_2 \square \ldots\ldots \square C_n \rightarrow S_n]\{P^n\}
\end{array}$$

Figure 6: Mechanisable Proof System

The proof system presented in figure 6 is mechanisable. Its equivalence to the proof system introduced above is shown in [4] where this new proof system is developed in a series of steps. The consequence rule is removed; the assignment axiom establishes as post-condition for pre-condition R, the set of properties directly deducable from R; a property has the form $x \notin \overline{y}$. For the assignment $y := exp(x_1, x_2, .., x_n)$, with pre-condition R, the post-condition property set is denoted as $T_y(R)$ and is defined as follows:

$$T_y(R) \mathrel{\hat{=}} \{ \ a \notin \overline{b} \mid a \notin \overline{b} \wedge b \neq y \ \} \bigcup R \cap \{ \ c \notin \overline{y} \mid c \notin \overline{x_i} \cup \mathit{val(indirect)} \wedge c \neq x_i \ \}$$

This means that all properties of the form $a \notin \overline{z}$ which are valid in R, are also valid after the command (for $z \neq y$), since they receive no flow of information. In the post-condition, properties of the form $(a \notin \overline{y})$ are not valid, even if they are valid in R, whenever a is in $\overline{x_i}$ or *indirect*; recall that these latter sets are added to $\overline{y}$ in the assignment semantics. The $\sqcup$ operator of the semantics is the property intersection operator:

$$(\ (x \notin \overline{y}) \wedge (z \notin \overline{t}) \ ) \sqcup (\ (x \notin \overline{t}) \wedge (z \notin \overline{t}) = (z \notin \overline{t})$$

Finally, the repetitive instruction is analyzed iteratively, until a fixed point is reached.

This "deterministic" semantics is used as the basis of a graph-based algorithm. Since all properties possess the form $x \notin \overline{y}$, the flow security state of a program can be represented by a directed graph. The nodes of the graph are the variables at each point of the program: thus, node x_i represents the variable x as it appears in instruction numbered i. An arc between two nodes represents an information flow; $(x_i \longrightarrow y_j)$ signifies that a flow of information occurs from x at program point i, to variable j at program point j. When no path exists between two variables, no information flow path exists. Each instruction parsed adds arcs to the program security graph.

As an example, consider the decryption program of figure 7. It accepts as inputs *cipher*, the cipher-text to be decrypted, a user's decryption *key*, and *unit*, the cost of decrypting a single character. The output *clear* goes to the user; the output *charge* goes to the user and the system administrator. A security requirement of this module might be that the system administrator does not receive an information flow form the user's clear-text *clear*. The resulting graph of this program is shown in figure 8; note the absense of a path from the *clear* variable to any instance of the *charge* variable. The program is thus secure since the administrator can infer nothing about the user's clear-text from the charge output.

```
var: $i$, $charge$, $key$, $unit$;
array: $clear$, $cipher$;
$cipher$ := ≺ message to be decrypted ≻;
$unit$ := ≺ unit rate constant ≻;
($p_1$,$charge$ := $unit$);
$i$ := 0;
($p_2$,*[ $cipher[i]$ ≠ null_constant →
          ($p_3$,($p_4$,[ encrypted($cipher[i]$) → ($p_5$,($p_6$,$clear[i]$ := D($cipher[i]$, $key$));
                                                          ($p_7$,$charge$ := $charge$ + 2*$unit$));
                   □ not encrypted($cipher[i]$) → ($p_8$,($p_9$,$clear[i]$ := $cipher[i]$);
                                                          ($p_{10}$,$charge$ := $charge$ + $unit$));
                   ]);
                   ($p_{11}$,$i$ := $i$ + 1))
])
```

Figure 7: Decryption Program

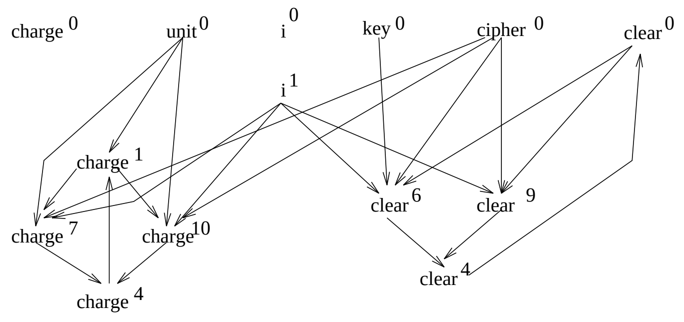

Figure 8: Graph generated for decryption program

4 Discussion

The approach outlined in this paper is based on Denning's work on information flow analysis of sequential programs [7]. Different flow analysis techniques for parallel programs are presented by Andrews & Reitman [1] and Mizuno & Oldhoeft [12]. A detailed analysis of this work is found in [3, 5].

The basic problem being addressed by this paper is that most systems use public domain software and that security violations can occur not only by illegally reading information, but by inferring the value of confidential information from other information using knowledge of the program text. The approach taken to tackle this problem is a code analysis one, where the semantics of information flow for a parallel programming language are defined, and an axiomatic proof system developed from this semantics. This proof system enables the security of parallel programmed systems to be verified by validating predicates on the information flows between variables that are inserted into the program text.

Other security issues that are also vital for future distributed computer systems include authentication, audit and intrusion tolerance. The problem being addressed in this paper is the basic security question - confidentiality of information stored. This question has still not being satisfactorily treated in computer system. Without a rigorous treatment of this question, these other mentioned mechanisms become superfluous.

The ideas put forward in this text are developed in [3] and [4]. Open areas of work include extending the compile-time analysis, catering for other programmed structures such as exceptions and also, proofs by composition. The latter is a facility for proving the security of a system by combining the proofs of its individual components.

References

[1] 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.

[2] 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.

[3] Banâtre (JP.), Bryce (C.), Information Flow Control in a Parallel Language Framework, in Proceedings of the 6th IEEE Workshop on the Foundations of Computer Security, Franconia, N.H., USA, June 1993, pages 46-61.

[4] Banâtre (JP.), Bryce (C.), Le Métayer (D.), Compile-time Analysis of Information Flow in Sequential Programs, in Proceedings of the 3rd European Symposium on Computer Security Research, Brighton, U.K., November 1994, pages 46-61.

[5] Bryce (C.), Étude et mise en œuvre des propriétés de sécurité, Rennes University, France, 1994.

[6] Cohen (L.), Information Transmission in Computational Systems, in Proceedings of the 6th Symposium on Operating System Principles, Texas, 1977, pages 133-139.

[7] Denning (D.), A Lattice Model of Secure Information Flow, in Communications of the ACM, 19 (5), May 1976, pages 236-243.

[8] Department of Defense, Trusted Computer System Evaluation Criteria, August 1983.

[9] Harrison (M.A.), Ruzzo (W.L.), Ullman (J.D.), Protection in Operating Systems, in Communications of the ACM, 19 (8), August 1976, pages 461-471.

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

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

[12] 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.

[13] Plotkin (G.D.), Structural Operational Semantics, in Lecture Notes, DAIMI FN-19, Aarhus University, Denmark, 1981.

[14] Reitman (R.), Information Flow in Parallel Programs, PhD Thesis, Cornell University, USA, 1978.

[15] Sandhu (R.), Lattice-Based Enforcement of Chinese Walls, in Computers and Security, 11 (1992), pages 753-763.