Available online at www.sciencedirect.com

SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 99 (2004) 87–110

www.elsevier.com/locate/entcs

Control Flow Analysis for the Applied π–calculus

Roberto Zunino [1,2]

Dipartimento di Informatica, Università di Pisa, Italy

Abstract

We define a control flow analysis for the applied π–calculus and we prove it correct with respect to the operational semantics. Then, we apply our analysis to verify a simple security property.

Keywords: static analysis, control flow analysis, applied pi calculus, security

1 Introduction

The importance of mobile calculi is steadily growing. Often, when dealing with networked environments and distributed systems, mobile calculi offer a good approach with a solid formal ground for designing new systems and protocols, as well as for studying existing ones. Mobile calculi are not meant to be full–featured programming languages for the implementation of systems; rather, they provide a core language for system specification that also facilitates reasoning about the properties of systems.

In many cases, the system is designed to work in an unsafe (and possibly hostile) environment where security is the main concern and cryptography is commonly used to prevent disclosure of secret data and to ensure data authenticity and integrity. However, using a good cryptosystem is not enough to guarantee that a system behaves in a secure manner: in fact, even with

[1] Partially supported by the EU project DEGAS and the MIUR project MEFISTO
[2] Email: zunino@di.unipi.it

1571-0661 © 2004 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2004.02.004

unbreakable cryptography, there is still room for attacks (such as replay and man–in–the–middle attacks). Under those circumstances, formal proofs of correctness are often required. However, manually proving the correctness of a large system can be a tedious, error–prone and expensive task.

Automatic static analysis techniques can dramatically reduce the cost of the formal proofs. Type systems[3] and control flow analysis have been defined for many mobile calculi as the π–calculus and the Spi calculus [3]. Examples of control flow analysis for the π–calculus and the Spi calculus can be found in [5,4,7,6].

The applied π–calculus has been introduced by Martín Abadi and Cédric Fournet in [2]. Its main advantage over other calculi lies in the fact that it does not use a fixed set of primitive operators (as *encrypt*, *decrypt*, *sign*, etc.) but follows a more general approach using an arbitrary set of terms and an equivalence relation that identifies semantically equivalent terms. In [2] it is shown how this set can be chosen for handling both symmetric and asymmetric cryptography, key exchange, message authentication codes and other security–related primitives. In spite of this flexibility, many general results can still be proved to hold for every choice of the set above. The paper in [10] also describes some applications of the calculus.

In this work we deal with a slight variant of the applied π–calculus. The syntax and semantics are basically the same of [2], the only significant change being that substitution variables can be introduced only at certain points, as we shall see in Section 3. We offer a technique for statically analyzing processes via control flow analysis. Then we prove our analysis correct with respect to the operational semantics. These are our main results and, to the best of our knowledge, this is the first proposal for a static analysis of the applied π–calculus.

More in detail, in Section 2 we define the syntax of the applied π–calculus. Terms, plain processes and extended processes are described along with their intuitive meaning. The operational semantics is discussed in Section 3: we first introduce some auxiliary relations (α–conversion, structural equivalence and the $>>$ reduction) and then we define the semantics of processes as a labeled transition system (the commitment relation). In Section 4 we introduce a control flow analysis for the applied π–calculus. In Section 5 we prove the *correctness* of our analysis (i.e. we show that the analysis agrees with the semantics) by establishing subject reduction. We deal with the existence of the minimum solution and we discuss some issues about its construction. Finally, in Section 6, we show how our analysis can be used to prove security properties.

[3] We shall not discuss in any detail the various type systems proposed for process calculi; we only cite here [1].

2 The Applied π–calculus

In this section we survey the applied π–calculus. We introduce its syntax and we intuitively describe its semantics, which we shall define in Section 3. Our definition of the operational semantics and more deeply that of the static analysis benefit from some changes to the original syntax. This is why many of the definitions presented here may appear more complex than strictly necessary.

2.1 Syntax

We use the following sets:

- $\mathcal{N} = \{n_i^{[j]} \mid i, j \in \mathbb{N}\}$ is the denumerable set of *names*;
- $\mathcal{L} = \{l_i \mid i \in \mathbb{N}\}$ is the denumerable set of *labels*;
- $\mathcal{X} = \{x_i^{[j]} \mid i, j \in \mathbb{N}\}$ is the denumerable set of *input variables*;
- $\mathcal{Z} = \{z_{l_i}^{[j]} \mid j \in \mathbb{N}, l_i \in \mathcal{L}\}$ is the denumerable set of *substitution variables*

where all the $n_i^{[j]}$, l_i, $x_i^{[j]}$, $z_{l_i}^{[j]}$ are assumed distinct. Intuitively, a name represents a communication channel which can be used to send and receive data of the form of *terms* (which we will define shortly); a label is used to mark a point in the program [4]; an input variable is a placeholder for received values (e.g. "read x from channel n_i and send x to channel n_j"); a substitution variable is used during communication to convey data. Our use of substitution variables, as we will see, is more restricted than in the applied π–calculus as defined in [2].

To simplify the notation, we simply write n, l, x, z_l (or n_i, x_i, z_{l_i}) where all their occurrences have the same indexes. For instance, we write $\mathsf{fst}(\mathsf{cons}(n_i, n_j)) = n_i$ instead of $\mathsf{fst}(\mathsf{cons}(n_i^{[k]}, n_j^{[m]})) = n_i^{[k]}$ and $\{n/x\}$ instead of $\{n_i^{[k]}/x_j^{[m]}\}$.

Accordingly, in inference rules or in propositions involving names, input variables and substitution variables, we sometimes write u to denote an element of $\mathcal{N}$, $\mathcal{X}$ or $\mathcal{Z}$ to avoid being too verbose. We now give the definition of terms.

Definition 2.1 Given a signature Σ consisting in a set of function symbols,

[4] This will be useful for static analysis. Moreover, labeling is performed automatically and labels are inserted in the internal representation of the process.

each with its own arity, *terms* are defined by the following grammar:

$M :=$	n	names
	x	input variables
	z_l	substitution variables
	$\mathsf{f}(M_1, \ldots, M_m)$	function application

where $\mathsf{f} \in \Sigma$, $n \in \mathcal{N}$, $x \in \mathcal{X}$, $z_l \in \mathcal{Z}$, $l \in \mathcal{L}$ and m is the arity of f.

We adopt the common convention that constants are functions of arity 0, thus including them into terms. Examples of terms are $M_1 = \mathsf{encrypt}(n_i, n_j)$, $M_2 = \mathsf{fst}(\mathsf{cons}(n, x))$ and $M_3 = n$ where $\mathsf{encrypt}$, fst and cons are function symbols. According to the intended meaning of cons and fst, the two terms M_2 and M_3 should be equivalent: we shall address this point when we will define the semantics. Since names are terms, they have both the roles of communication channels and of data: this aspect is crucial for mobility. Moreover, names are also used to denote cryptographic keys and secrets: for instance, encryption of a term M with a symmetric key n could be represented as $\mathsf{encrypt}(M, n)$. Similarly, $\mathsf{decrypt}(\mathsf{encrypt}(M, n), n)$ could be used to model decryption. Asymmetric cryptography can be modeled as well: $\mathsf{pri}(n)$ and $\mathsf{pub}(n)$ can be used to model corresponding keys so that the term $\mathsf{decrypt}(\mathsf{encrypt}(M, \mathsf{pub}(n_i)), \mathsf{pri}(n_j))$ is equivalent to M if and only if $i = j$.

A *ground* term is a term without variables (of any kind).

Definition 2.2 *Plain processes* are recursively defined by the following grammar:

$P :=$	0	nil
	$P_1 \mid P_2$	parallel composition
	$!P$	replication
	$\nu n.P$	name restriction
	if $M_1 = M_2$ *then* P_1 *else* P_2	conditional
	$M(x).P$	input
	$\overline{M_1}\langle M_2 \rangle^l.P$	output

Intuitively, 0 is the process that does nothing; $P \mid Q$ runs P and Q concurrently; $!P$ represents an unlimited number of instances of P running concurrently; restriction defines the scope of the name, actually creating a new local name; a conditional compares two ground terms for equality; input receives some value from the channel M (the *subject*), binds it to the variable x (the *object*) and then behaves as P (the *continuation*); output sends the value M_2 (the *object*) to the channel M_1 (the *subject*) and then behaves as P (the *continuation*).

The term used as the subject in input and output processes should evaluate to a name at run–time; otherwise they behave as 0. Also, note that labels are used to mark outputs.

For example, the following are processes:

$$\begin{aligned}
A_1 &= n_1(x).\overline{\mathsf{fst}(x)}\langle \mathsf{snd}(x)\rangle^{l_1}.0 \mid \overline{n_1}\langle \mathsf{cons}(n_2, n_3)\rangle^{l_2}.0\\
A_2 &= \nu n_1.\Big(\nu n_2.\overline{n_0}\langle \mathsf{sign}(n_2, \mathsf{pri}(n_1))\rangle^{l_0}.0 \mid\\
&\qquad n_0(x).\text{if } \mathsf{check}(x, \mathsf{pub}(n_1)) = \mathsf{ok}\\
&\qquad \text{then } \overline{n_0}\langle \mathsf{removesign}(x)\rangle^{l_1}.0\\
&\qquad \text{else } \overline{n_0}\langle \mathsf{error}\rangle^{l_2}.0\Big)
\end{aligned}$$

Accordingly to the intuitive semantics, in the first example A_1, $\mathsf{fst}(x)$ at run-time evaluates to a name (n_2). The second example A_2 shows a process that creates a new name (n_2), signs it with a private key $\mathsf{pri}(n_1)$ and sends it. Another process receives it, checks the signature and uses n_2. Note that for A_2 to work, the term $\mathsf{check}(\mathsf{sign}(n_2, \mathsf{pri}(n_1)), \mathsf{pub}(n_1))$ should evaluate to ok and $\mathsf{removesign}(\mathsf{sign}(n_2, \mathsf{pri}(n_1)))$ should evaluate to n_2.

In order to simplify the semantics, we extend the syntax of plain processes to include *substitutions*.

Definition 2.3 *Extended processes* are recursively defined by the following grammar:

$$\begin{array}{lll}
A := & P & \text{plain process}\\
& A_1 \mid A_2 & \text{parallel composition}\\
& \nu n.A & \text{name restriction}\\
& \nu z_l.A & \text{variable restriction}\\
& \{M/z_l\} & \text{substitution}
\end{array}$$

A substitution process is used to bind some substitution variable to some term. For example

$$\nu z_l.(\{n/z_l\} \mid \overline{z_l}\langle z_l\rangle^l.0)$$

can be reduced to $\overline{n}\langle n\rangle^l.0$. We will return on the issue of introducing and eliminating substitution variables in Section 3.

We write $\mathsf{bound}(A)$ for the set of all the $u \in \mathcal{N} \cup \mathcal{X} \cup \mathcal{Z}$ that occur in A under a restriction or as the object of an input. We define $\mathsf{sym}(A)$ to be the set of all the $u \in \mathcal{N} \cup \mathcal{X} \cup \mathcal{Z}$ that occur in A and $\mathsf{sym}(M)$ to be the set of all the $u \in \mathcal{N} \cup \mathcal{X} \cup \mathcal{Z}$ that occur in M. We write $\mathsf{free}(A)$ for $\mathsf{sym}(A) \setminus \mathsf{bound}(A)$. We also use $\mathsf{vars}(M)$ for $\mathsf{sym}(M) \cap (\mathcal{X} \cup \mathcal{Z})$.

As usual, a context is a process (or a term) with a hole. There are contexts of type process with an hole of type process (as $A_1 \mid [-]$), contexts of type process with an hole of type term (as $\overline{n}\langle[-]\rangle.0$) and contexts of type term with an hole of type term (as $\mathsf{cons}(n, [-])$).

If $\mathsf{sym}(M) \cap \mathsf{bound}(A) = \emptyset$, we write $A\{M/z_l\}$ for the replacement of all free occurrences of z_l with M. Of course, this constraint is always satisfied if we suitably α–convert A before substituting.

3 Operational Semantics

In this section we define a formal semantics of the applied π–calculus. We point out the differences with the semantics given in [2] and describe the rationale for those differences.

The semantics is defined by steps. First, α–conversion ($=_\alpha$), structural equivalence ($\equiv$) and reduction($>>$) are introduced. Secondly, the commitment relation ($\xrightarrow{\alpha}$) is used to define the actual semantics.

3.1 α–Conversion

As usual, α–conversion is a kind of variable renaming that is sometimes necessary: for instance, in the process

$$\nu n_1^{[0]}.\overline{n_0}\langle n_1^{[0]}\rangle^{l_0}.0 \mid \nu n_1^{[0]}.\overline{n_0}\langle n_1^{[0]}\rangle^{l_1}.0 \mid n_0(x_0).n_0(x_1).P$$

the values assumed by x_0 and x_1 are originated by two different ν and therefore should be different. Because of this, the operational semantics must allow communication only if one $n_1^{[0]}$ has been renamed. For example we could rewrite the above process into

$$\nu n_1^{[0]}.\overline{n_0}\langle n_1^{[0]}\rangle^{l_0}.0 \mid \nu n_1^{[1]}.\overline{n_0}\langle n_1^{[1]}\rangle^{l_1}.0 \mid n_0(x_0).n_0(x_1).P$$

allowing communication.

Definition 3.1 We define *α-conversion* ($=_\alpha$) as the minimum equivalence relation on extended processes that is closed under contexts and such that

$$\begin{array}{lll} \text{Res} & \nu u_k^{[i]}.A =_\alpha \nu u_k^{[j]}.A\{u_k^{[j]}/u_k^{[i]}\} & \text{if } u_k^{[j]} \notin \mathsf{sym}(A) \\ \text{In} & M(x_k^{[i]}).P =_\alpha M(x_k^{[j]}).P\{x_k^{[j]}/x_k^{[i]}\} & \text{if } x_k^{[j]} \notin \mathsf{sym}(P) \end{array}$$

Note that this form of α-conversion never changes the subscript index k. We have chosen to only rename the superscript so that, when dealing with static analysis, we can trace an α–converted name (or variable) to its originating restriction. This is possible if we initially choose unique subscripts for names and variables, which we can always do before performing the analysis.

Moreover, we choose $u_k^{[0]}$ to be the *canonical* name (or *canonical variable*) for all the names (or variables) of the form $u_k^{[i]}$. Similarly, we say that a term is *canonical* if only canonical names and variables occur in it. This helps when considering names or variables up to α–conversion.

3.2 Structural Equivalence

Syntax often imposes some unwanted structure on processes: for instance, while $A_1 \mid A_2$ and $A_2 \mid A_1$ are syntactically different processes, we want to identify them semantically. Moreover, we also want to identify some terms like $\mathsf{fst}(\mathsf{cons}(n_1, n2))$ and n_1 under the standard intuition for fst and cons. For these purposes we use *structural equivalence.*

We consider an equivalence relation on ground terms ($\Sigma \vdash M_1 = M_2$) that is closed under application of contexts (of type term with an hole of type term). This relation will be used to identify processes of the form $A[M_1]$ and $A[M_2]$ where $A[-]$ is a context and $\Sigma \vdash M_1 = M_2$ (rule Context below). We sometimes write $\Sigma \not\vdash M_1 = M_2$ for $\neg(\Sigma \vdash M_1 = M_2)$.

Definition 3.2 *Structural equivalence* ($\equiv$) is the minimum equivalence relation on extended processes that is preserved by contexts and such that

Alpha	$A_1 \equiv A_2$	if $A_1 =_\alpha A_2$
Par-0	$A \equiv A \mid 0$	
Par-Assoc	$A_1 \mid (A_2 \mid A_3) \equiv (A_1 \mid A_2) \mid A_3$	
Par-Comm	$A_1 \mid A_2 \equiv A_2 \mid A_1$	
Repl	$!P \equiv P \mid !P$	
Res-0	$\nu n.0 \equiv 0$	
Res-Comm	$\nu u_i.\nu u_j.A \equiv \nu u_j.\nu u_i.A$	
Res-Scope	$A_1 \mid \nu u.A_2 \equiv \nu u.(A_1 \mid A_2)$	if $u \notin \mathsf{free}(A_1)$
Subst	$\overline{M_1}\langle M_2\rangle^l.P \equiv \nu z_l.(\overline{M_1}\langle z_l\rangle^l.P \mid \{M_2/z_l\})$ if $z_l \notin \mathsf{sym}(M_1) \cup \mathsf{sym}(M_2) \cup \mathsf{free}(P)$	
Context	$A[M_1] \equiv A[M_2]$	if $\Sigma \vdash M_1 = M_2$

A few points are noteworthy: Repl rule, for instance, is all we need to define the semantics of replication. Res-Scope "moves" restrictions: note that in some cases we have to α–convert u to enlarge its scope. This is mainly used in the following case, assuming $u \notin \mathsf{free}(n(x).P)$:

$$n(x).P_1 \mid \nu u.(P_3 \mid \overline{n}\langle M\rangle.P_2) \equiv \nu u.(n(x).P_1 \mid \overline{n}\langle M\rangle.P_2 \mid P_3)$$

The processes performing input and output are now side-by-side, so that they can communicate.

In the rule Context, $A[-]$ is a context of type process with a hole of type term.

The Subst rule is a special case of the more general rule

$$\text{General-Subst} \quad A\{M/x\} \equiv \nu x.(A \mid \{M/x\}) \quad \text{if } x \notin \mathsf{sym}(M)$$

that appears in [2], where there is no distinction between input variables and substitution variables. In fact, if we allowed General-Subst we could simplify both syntax and semantics: there would be no need of distinguishing between $\mathcal{X}$ and $\mathcal{Z}$ (thus using a single set of variables) and no need of the reduction relation (that we are going to define in Section 3.3). In spite of this, it would be much harder to define the control flow analysis. The difficulty lies in the fact that at run–time new variables can appear and that statically we cannot know anything about them. Unlike General-Subst, the rule Subst "remembers" which output generated the substitution: for instance, if the substitution $\{M/z_{l_{51}}\}$ is generated at run–time, we know that M was the object of the output whose label is l_{51}.

3.3 Reduction

Although we decided not to use General-Subst, we cannot simply discard it. While we do not want to introduce new variables (via General-Subst applied left-to-right), we should permit the elimination of variables and the application of substitution processes to adjacent processes (via General-Subst applied right-to-left). To this purpose, we introduce the following *reduction* relation.

Definition 3.3 *Reduction* ($>>$) is the minimum transitive relation that is closed under contexts and such that

$$\begin{array}{lll} \text{Equiv} & A_1 >> A_2 & \text{if } A_1 \equiv A_2 \\ \text{Res} & \nu z_l.\{M/z_l\} >> 0 & \\ \text{Subst} & \{M/z_l\} \mid A >> \{M/z_l\} \mid A\{M/z_l\} & \end{array}$$

We now compare reduction with structural equivalence "enriched" with the General-Subst rule. In order to do the comparison, we assume $\mathcal{X} = \mathcal{Z}$; i.e., we make no distinction between input and substitution variables. We define $\equiv_{GS}$ in the same fashion as $\equiv$ (Definition 3.2) but adding General-Subst to the rule set.

If A_1 and A_2 have no free substitution variables, the following property holds:

$$A_1 >> A_2 \implies A_1 \equiv_{GS} A_2$$

This property ensures that the reduction relation does not extend the semantics given in [2].

The converse implication

$$A_1 \equiv_{GS} A_2 \implies A_1 >> A_2$$

also holds as long as new substitution variables are not introduced in A_2. Under this hypothesis, $\equiv_{GS}$ is a conservative extension of $>>$.

3.4 Commitment

The commitment relation defines the actual semantics for extended processes, describing a labeled transition system. We write $A_1 \xrightarrow{\mu} A_2$ when the process A_1 performs the action μ and transforms itself into A_2. Actions μ can be inputs $(n(M))$, output $(\nu \vec{u} . \overline{n}\langle M\rangle)$ or silent moves (τ).

The input action is performed when the process reads something from the outside world, and similarly the outputs sends data to the outside; in the last case the object could contain some restricted variables and names that become "public" (i.e. unrestricted) because of the output: these *extruded* variables and names are shown in the action as the $\vec{u}$ vector (possibly empty). The τ actions are used to represent internal communication (that is, from one process to another).

For example the process $A = n(x).P_1 \mid \overline{n}\langle M\rangle^l.P_2$ can perform the following actions

$$\begin{aligned}
A &\xrightarrow{n(M_1)} P_1\{M_1/x\} \mid \overline{n}\langle M\rangle^l.P_2\\
A &\xrightarrow{\overline{n}\langle M\rangle} n(x).P_1 \mid P_2\\
A &\xrightarrow{\tau} P_1\{M/x\} \mid P_2
\end{aligned}$$

where the $\vec{u}$ in the second action is empty.

We extend free, bound and sym to actions: $\mathsf{free}(\mu)$ are all the u not under a $\nu \vec{u}$ that occur in μ; $\mathsf{bound}(\mu)$ are the restricted ones; $\mathsf{sym}(\mu) = \mathsf{free}(\mu) \cup \mathsf{bound}(\mu)$. We are now ready to define the commitment relation.

Definition 3.4 The commitment relation $A_1 \xrightarrow{\mu} A_2$ is defined by the following inference rules:

$$\textit{Comm} \qquad \overline{n}\langle u\rangle^l.P_1 \mid n(x).P_2 \xrightarrow{\tau} P_1 \mid P_2\{u/x\}$$

$$\textit{Then} \qquad \frac{\mathsf{vars}(M) = \emptyset}{\textit{if } M = M \textit{ then } P_1 \textit{ else } P_2 \xrightarrow{\tau} P_1}$$

$$\textit{Else} \qquad \frac{\mathsf{vars}(M_1) = \mathsf{vars}(M_2) = \emptyset \quad \Sigma \not\vdash M_1 = M_2}{\textit{if } M_1 = M_2 \textit{ then } P_1 \textit{ else } P_2 \xrightarrow{\tau} P_2}$$

$$\textit{In} \qquad n(x).P \xrightarrow{n(M)} P\{M/x\}$$

$$\textit{Out-Term} \qquad \overline{n}\langle M\rangle^l.P \xrightarrow{\overline{n}\langle M\rangle} P$$

$$\textit{Open-All} \qquad \frac{A_1 \xrightarrow{\nu\vec{u}.\overline{n}\langle M\rangle} A_2 \quad u_i \in \mathsf{free}(M) \setminus \{n, \vec{u}\}}{\nu u_i.A_1 \xrightarrow{\nu u_i,\vec{u}.\overline{n}\langle M\rangle} A_2}$$

$$\textit{Scope} \qquad \frac{A_1 \xrightarrow{\mu} A_2 \quad u \notin \mathsf{sym}(\mu)}{\nu u.A_1 \xrightarrow{\mu} \nu u.A_2}$$

$$\textit{Par} \qquad \frac{A_1 \xrightarrow{\mu} A_2 \quad \mathsf{bound}(\mu) \cap \mathsf{free}(A_3) = \emptyset}{A_1 \mid A_3 \xrightarrow{\mu} A_2 \mid A_3}$$

$$\textit{Struct} \qquad \frac{A_1 >> A_2 \quad A_2 \xrightarrow{\mu} A_3 \quad A_3 >> A_4}{A_1 \xrightarrow{\mu} A_4}$$

Note that for inter–process communication, there is no need for *extrusion.* It suffices to apply α–conversion and structural equivalence to enlarge the scope of the restricted variables so that they can be made adjacent. The rule Comm is rather simple: it only transmits variables and names. If we want to transmit a more complex term, we can use structural equivalence in the following way (assuming $z_l \notin \mathsf{sym}(P)$):

$$\begin{aligned}
& n_0(x).P \mid \overline{n_0}\langle \mathsf{cons}(n_1, n_2)\rangle^l.0 \\
\equiv\quad & n_0(x).P \mid \nu z_l.(\{\mathsf{cons}(n_1, n_2)/z_l\} \mid \overline{n_0}\langle z_l\rangle^l.0) \\
\equiv\quad & \nu z_l.(n_0(x).P \mid \{\mathsf{cons}(n_1, n_2)/z_l\} \mid \overline{n_0}\langle z_l\rangle^l.0) \\
\xrightarrow{\tau}\quad & \nu z_l.(P\{z_l/x\} \mid \{\mathsf{cons}(n_1, n_2)/z_l\} \mid 0) \\
>>\quad & \nu z_l.(P\{\mathsf{cons}(n_1, n_2)/x\} \mid \{\mathsf{cons}(n_1, n_2)/z_l\} \mid 0) \\
\equiv\quad & P\{\mathsf{cons}(n_1, n_2)/x\} \mid \nu z_l.\{\mathsf{cons}(n_1, n_2)/z_l\} \\
>>\quad & P\{\mathsf{cons}(n_1, n_2)/x\} \mid 0 \\
\equiv\quad & P\{\mathsf{cons}(n_1, n_2)/x\}
\end{aligned}$$

Applying Struct we obtain

$$n_0(x).P \mid \overline{n_0}\langle \mathsf{cons}(n_1, n_2)\rangle^l.0 \xrightarrow{\tau} P\{\mathsf{cons}(n_1, n_2)/x\}$$

There is no rule for 0 since it cannot do any action. Moreover, we note that a conditional is executed only when the terms are *ground* and otherwise it behaves as 0. Inputs and outputs also behave as 0 unless the subject is a name (or is reduced via $>>$ to a name).

Our operational semantics is *label insensitive*. In fact, the definition of the commitment relation never involve labels. Labels affect only the Subst rule in structural equivalence ($\equiv$) since the new substitution variable z_l is chosen to match the label.

4 Control Flow Analysis

In this section we define the *control flow analysis* (CFA) for the applied π–calculus. The main difference between this and previous analyses for other calculi (e.g. π) lies in the fact that we need to handle Σ–equivalence between terms and deal with substitution processes.

The purpose of CFA is to determine statically which parts of a program may be reached from which other parts and what values are passed between them. In the applied π–calculus, control flow is rather primitive because there are no equivalents for loops, function calls or recursion that are commonly used in some other paradigms. However, it is useful to analyze the values that are transmitted on channels and the values that are assumed by variables at run–time.

4.1 CFA for the Applied π–calculus

We let the set of values Val be the set of canonical ground terms, and use the following functions:

- $\rho : \mathcal{X} \cup \mathcal{Z} \to \mathcal{P}(Val)$ is the *abstract environment* that maps a variable to the set of all values that may be assumed by that variable;
- $\kappa : \mathcal{N} \to \mathcal{P}(Val)$ is the *abstract channel environment* that maps a channel (a name) to the set of all values that may be transmitted on that channel.

Thus, $\rho(x)$ represents an *upper bound* to the (set of) values that x can assume at run–time. Similarly, $\kappa(n)$ is an upper bound to the values that are transmitted over the channel n at run–time.

Instead, $\rho(z_l)$ has a twofold meaning. First, it is an upper bound to the values assumed by terms that occur as objects in outputs labeled with l. For example, in $\overline{n}\langle M\rangle^l.P$ we expect $\rho(z_l)$ to include all the possible run–time values of M. Second, $\rho(z_l)$ is an upper bound to the values assumed by terms in *substitutions* of z_l. For example, in $\{M/z_l\} \mid A$ we expect $\rho(z_l)$ to include all the possible run–time values of M. This double nature of $\rho(z_l)$ is related to the Subst rules of $\equiv$ and $>>$.

Below, we define when a pair (ρ, κ) is *acceptable* for an extended process A; that is, (ρ, κ) gives an actual upper bound to run–time semantics. Since the problem in this form is obviously undecidable, we must put strong constraints on (ρ, κ) to make *acceptability* statically computable. We will call the pairs (ρ, κ) that satisfy such constraints *CFA solutions* (for an extended process A).

After defining which pairs (ρ, κ) are CFA solutions, we must show that they actually agree with run-time semantics (i.e., that they are upper bounds): this *correctness* property is called *subject reduction* and is proved in Section 5.

The following are natural, basic constraints that CFA solutions must satisfy:

$$\alpha\text{–}inv \qquad \forall i,j,k \in \mathbb{N} : \begin{cases} \rho(x_k^{[i]}) = \rho(x_k^{[j]}) \\ \rho(z_{l_k}^{[i]}) = \rho(z_{l_k}^{[j]}) \\ \kappa(n_k^{[i]}) = \kappa(n_k^{[j]}) \end{cases}$$

$$\Sigma\text{–}clos \qquad \forall x \in \mathcal{X}, z_l \in \mathcal{Z}, n \in \mathcal{N}, M_1, M_2 \in Val : \Sigma \vdash M_1 = M_2 \implies \begin{cases} M_1 \in \rho(x) \implies M_2 \in \rho(x) \\ M_1 \in \rho(z_l) \implies M_2 \in \rho(z_l) \\ M_1 \in \kappa(n) \implies M_2 \in \kappa(n) \end{cases}$$

The α–inv constraint makes solutions stable under α–conversion while Σ–clos is used to identify Σ–equivalent terms, making $\rho(x), \rho(z_l)$ and $\kappa(n)$ closed under Σ–equivalence.

We now define the *abstract semantics for terms*, that is the set of canonical values that a term may assume at run–time. Since a term may contain variables, its values depend on ρ.

Definition 4.1 (Abstract Semantics for Terms)
If V is a set of values, we write V^Σ for its closure under Σ–equivalence. We define the abstract semantics of a term $[\![-]\!]_\rho$ by structural induction in the following way:

$$\begin{aligned}
[\![n_k^{[i]}]\!]_\rho &= \{n_k^{[0]}\}^\Sigma \\
[\![x_k^{[i]}]\!]_\rho &= \rho(x_k^{[0]}) \\
[\![z_{l_k}^{[i]}]\!]_\rho &= \rho(z_{l_k}^{[0]}) \\
[\![\mathsf{f}(M_1, \ldots, M_k)]\!]_\rho &= \{\mathsf{f}(m_1, \ldots, m_k) \mid \forall i \in [1..k] : m_i \in [\![M_i]\!]_\rho\}^\Sigma
\end{aligned}$$

Using the abstract semantics for terms, we are now ready to define the control flow analysis for the applied π–calculus.

Definition 4.2 (Control Flow Analysis)
A pair $(\rho, \kappa) \in (\mathcal{X} \cup \mathcal{Z} \to \mathcal{P}(Val)) \times (\mathcal{N} \to \mathcal{P}(Val))$ that satisfies the α–inv and Σ–clos constraints is a CFA solution for an extended process A ($(\rho, \kappa) \models A$)

if and only if it satisfies the following clauses.

$$
\begin{array}{llll}
Nil & (\rho,\kappa) \models 0 & \Leftrightarrow & True \\
Par & (\rho,\kappa) \models A_1 \mid A_2 & \Leftrightarrow & (\rho,\kappa) \models A_1 \wedge (\rho,\kappa) \models A_2 \\
Nres & (\rho,\kappa) \models \nu n.A & \Leftrightarrow & (\rho,\kappa) \models A \\
Vres & (\rho,\kappa) \models \nu z_l.A & \Leftrightarrow & (\rho,\kappa) \models A \\
Subst & (\rho,\kappa) \models \{M/z_l\} & \Leftrightarrow & \rho(z_l) \supseteq [\![M]\!]_\rho \\
Repl & (\rho,\kappa) \models !P & \Leftrightarrow & (\rho,\kappa) \models P \\
If & (\rho,\kappa) \models \begin{array}{l} \mathit{if}\ M_1 = M_2 \\ \mathit{then}\ P_1\ \mathit{else}\ P_2 \end{array} & \Leftrightarrow & \begin{array}{l} ([\![M_1]\!]_\rho \cap [\![M_2]\!]_\rho = \emptyset \vee (\rho,\kappa) \models P_1) \wedge \\ (\rho,\kappa) \models P_2 \end{array} \\
In & (\rho,\kappa) \models M(x).P & \Leftrightarrow & \forall n \in [\![M]\!]_\rho : \rho(x) \supseteq \kappa(n) \wedge (\rho,\kappa) \models P \\
Out & (\rho,\kappa) \models \overline{M_1}\langle M_2\rangle^l.P & \Leftrightarrow & \begin{array}{l} \forall n \in [\![M_1]\!]_\rho : \kappa(n) \supseteq \rho(z_l) \wedge \\ \rho(z_l) \supseteq [\![M_2]\!]_\rho \wedge (\rho,\kappa) \models P \end{array}
\end{array}
$$

Any (ρ,κ) is a solution for 0. A solution for a parallel composition of two processes is a solution for each of them. Restriction of names and (substitution) variables does not affect solutions, as well as replication. A substitution requires that $\rho(z_l)$ is an upper bound to the possible values of the term M. A solution for a conditional must always be a solution for the "else" branch. If the abstract semantics of M_1 and M_2 are disjoint, we infer that the "then" branch is never executed and we put no further constraints on (ρ,κ). If this is not the case, we require that (ρ,κ) be valid for both branches. A solution for an input process must

(i) ensure that $\rho(x)$ is an upper bound to all values that are transmitted on all channels which are the result of the abstract evaluation of M and

(ii) be a solution for the input continuation P.

An output requires that

(i) $\rho(z_l)$ is an upper bound to the abstract evaluation of M_2 and

(ii) for every channel n that is the result of the abstract evaluation of M_1, $\kappa(n)$ contains $\rho(z_l)$ (and therefore it contains every value of M_2) and

(iii) the solution is also a solution for the output continuation P.

The definition above is by no means the only possible one. The Appendix briefly surveys some alternatives. Indeed, there is huge room for further improvements to our CFA. However, proving correctness for more precise CFA can be very hard. For instance, to prove subject reduction for the last CFA variant for conditionals given in the Appendix, one has to consider all the execution of the process, i.e. all the sequence of commitment steps.

5 Subject Reduction

In this section we show the correctness the CFA of Section 4 with respect to the formal semantics of Section 3 by proving a *subject reduction* result. Most proofs are by structural induction or rule induction, and are quite straightforward. In the case, we simply sketch the proof.

We also deal with the existence of the minimum CFA solution and its construction.

5.1 Preliminary Lemmata

The following lemma is useful when dealing with substitutions and α–conversion.

Lemma 5.1 *Substitution Lemma*

$$u \in \mathcal{X} \cup \mathcal{Z} \wedge \rho(u) \supseteq [\![M_1]\!]_\rho \implies [\![M_2]\!]_\rho \supseteq [\![M_2\{M_1/u\}]\!]_\rho \tag{1}$$

$$u \in \mathcal{X} \cup \mathcal{Z} \wedge (\rho,\kappa) \models A \wedge \rho(u) \supseteq [\![M]\!]_\rho \implies (\rho,\kappa) \models A\{M/u\} \tag{2}$$

$$n_k^{[j]} \notin \mathsf{sym}(A) \wedge (\rho,\kappa) \models A \implies (\rho,\kappa) \models A\{n_k^{[j]}/n_k^{[i]}\} \tag{3}$$

Proof. It suffices to apply structural induction on M and A. In order to show (3) we also use α–inv. □

Under suitable hypotheses, the context lemma stated below allows us to replace a subprocess or a subterm of a given extended process A without invalidating CFA solutions.

Lemma 5.2 *Context Lemma*
Let $A[-]$ be a context. We have

$$(\rho,\kappa) \models A[A_1] \wedge ((\rho,\kappa) \models A_1 \Rightarrow (\rho,\kappa) \models A_2) \implies (\rho,\kappa) \models A[A_2] \tag{4}$$

$$(\rho,\kappa) \models A[M_1] \wedge \Sigma \vdash M_1 = M_2 \implies (\rho,\kappa) \models A[M_2] \tag{5}$$

Proof. By structural induction on the context $A[-]$, also using Σ–clos for (5). □

Our CFA solutions are also stable under $=_\alpha$, $\equiv$, $>>$.

Lemma 5.3 *Stability Lemma*

$$(\rho,\kappa) \models A_1 \wedge A_1 =_\alpha A_2 \implies (\rho,\kappa) \models A_2 \tag{6}$$
$$(\rho,\kappa) \models A_1 \wedge A_1 \equiv A_2 \implies (\rho,\kappa) \models A_2 \tag{7}$$
$$(\rho,\kappa) \models A_1 \wedge A_1 >> A_2 \implies (\rho,\kappa) \models A_2 \tag{8}$$

Proof. The proof is by rule induction on the definitions of the relations $=_\alpha$, $\equiv$, and $>>$; we use here both the Substitution Lemma and the Context Lemma. Most cases are straightforward: the most interesting one is that of (7) when applying the Subst rule left-to-right.

$$\overline{M_1}\langle M_2\rangle^l.P \quad \equiv \quad \nu z_l.(\overline{M_1}\langle z_l\rangle^l.P \mid \{M_2/z_l\})$$

In this case, the label l plays a crucial rôle. The CFA definition for the output process in fact foresees the introduction of the variable z_l by requiring that $\kappa(n) \supseteq \rho(z_l)$ and $\rho(z_l) \supseteq [\![M_2]\!]_\rho$. This allows us to show that (ρ,κ) is also a solution for the right hand–side process. □

5.2 Subject Reduction

This is our main result that proves the semantic correctness of our CFA. It says that at run–time A performs a sequence of actions μ_i that have been "statically predicted" by the CFA solution (ρ,κ), provided that the inputs from the environment agree with the abstract channel environment[5] κ.

For proving subject reduction, it suffices to show the correctness for a single step $A_1 \xrightarrow{\mu} A_2$ and to show that CFA solutions are preserved by the commitment relation, allowing induction on steps.

Theorem 5.4 *Subject Reduction*

$$(\rho,\kappa) \models A_1 \wedge A_1 \xrightarrow{\mu} A_2 \implies$$
$$\begin{cases} \mu = \tau & \implies (\rho,\kappa) \models A_2 \\ \mu = n(M) \wedge \kappa(n) \supseteq [\![M]\!]_\rho & \implies (\rho,\kappa) \models A_2 \\ \mu = \nu\vec{u}.\overline{n}\langle M\rangle & \implies \kappa(n) \supseteq [\![M]\!]_\rho \wedge (\rho,\kappa) \models A_2 \end{cases}$$

[5] This hypothesis plays a crucial role when dealing with security properties. In Section 6 we show why it is important.

Proof. The proof is by rule induction on the definition of the commitment relation. □

5.3 Existence of Solutions

The procedure induced by Definition 4.2 tests which pairs (ρ, κ) are acceptable solutions for a process A. In this section we show that, given a process A, the minimum CFA solution exists, according to the following ordering:

Definition 5.5 We write Sol for the set of the pairs (ρ, κ) that satisfy α–inv and Σ–clos. We partially order the set Sol in the following way:

$$(\rho, \kappa) \sqsupseteq (\rho', \kappa') \iff \begin{cases} \forall u \in \mathcal{X} \cup \mathcal{Z} : \rho(u) \supseteq \rho'(u) \\ \forall n \in \mathcal{N} : \kappa(n) \supseteq \kappa'(n) \end{cases}$$

The minimum CFA solution provides the tightest bound for values that can be assumed by variables and for values that can be sent over channels at run–time.

We write $\bigsqcap \mathsf{S}$ for the greatest lower bound of the set $\mathsf{S} \subseteq Sol$, and $\bigsqcup \mathsf{S}$ for its least upper bound. It is easy to see that $(Sol, \sqsupseteq)$ is a complete lattice, since the ordering is pointwise and the properties α–inv and Σ–clos are preserved by $\bigsqcup$ and $\bigsqcap$. We write $(\bot, \bot)$ for $\bigsqcup \emptyset$.

The existence of the minimum solution follows from the fact that solutions form a *Moore family*:

Definition 5.6 Let S be a subset of Sol. S is a *Moore family* if and only if $\forall \mathsf{Z} \subseteq \mathsf{S} : \bigsqcap \mathsf{Z} \in \mathsf{S}$.

The following theorem proves that the set S of the CFA solutions for a process A is a Moore family. Therefore, $\bigsqcap \mathsf{S}$ is the minimum CFA solution for A.

Theorem 5.7 (Existence of CFA Solutions)
Let A be an extended process. The set

$$\mathsf{S} = \{(\rho, \kappa) \mid (\rho, \kappa) \models A\}$$

is a Moore family.

Proof.
Structural induction on A suffices to prove the following property:

$$\forall (\overline{\rho}, \overline{\kappa}) \in Sol : \{(\rho, \kappa) \mid (\rho, \kappa) \models A \wedge (\rho, \kappa) \sqsupseteq (\overline{\rho}, \overline{\kappa})\} \text{ is a Moore family} \quad (*)$$

The theorem then follows by letting $(\overline{\rho}, \overline{\kappa}) = (\bot, \bot)$.

Since $(*)$ is a standard result for CFA solutions, the proof is very similar to those present in the literature. See e.g. [6] where the property above is proved for the π–calculus. We omit this proof. $\square$

5.4 Construction of CFA Solutions

We now briefly discuss a procedure which computes the minimum CFA solution for a process A.

The standard approach is to derive from A a set of *constraints* over the domain D of all the pairs (ρ, κ) in such a way that the constraints are satisfied by (ρ, κ) if and only if $(\rho, \kappa) \models A$. Then a constraint–solving algorithm is applied, thus computing the minimum element D satisfying the constraints (see, e.g. [6,8]).

The generation of constraints from our CFA in Definition 4.2 can be performed in a rather straightforward way by induction on the structure of A. Nevertheless, some points are noteworthy:

(i) The sets $\mathcal{N}, \mathcal{X}$ and $\mathcal{Z}$ are infinite. This, however, poses no problems since we can restrict our procedure to deal only with $\mathsf{sym}(A)$ (i.e., the names and variables occurring in A) which is a finite set.

(ii) Our constraints are over the domain

$$D = (\mathcal{X} \cup \mathcal{Z} \to \mathcal{P}(Val)) \times (\mathcal{N} \to \mathcal{P}(Val))$$

To make constraint–solving computable, we need a finite representation for its elements. This basically means we need a finite representation for sets of values that are closed under Σ–equivalence such as $[\![M]\!]_\rho$, $\rho(z_l)$ and $\kappa(n)$.

(iii) Such a representation must have an operation to join sets of values: for instance, we need to reduce the constraint set $\{\kappa(n) \supseteq [\![M_1]\!]_\rho, \kappa(n) \supseteq [\![M_2]\!]_\rho\}$ into $\{\kappa(n) \supseteq \mathsf{V}\}$ for suitable V, the join of $[\![M_1]\!]_\rho$ and $[\![M_2]\!]_\rho$.

(iv) Disjointness of two value sets must be decidable in order to handle the constraints generated by the If case.

Below, we briefly discuss some possible cases.

First, if Σ–equivalence is not decidable, there is no algorithm for solving constraints. Moreover, even if Σ–equivalence is decidable, we foresee difficulties if it does not admit normal forms. Also, the computational complexity of the constraint–solving algorithm is deeply affected by Σ–equivalence. We presently do not know when Σ–equivalence makes it possible to efficiently construct the minimum CFA solution. This is one of the main points for future research.

We now consider the simplest case, i.e. when Σ–equivalence is the identity relation and we use the free term algebra. This is the case, e.g. for the Spi–calculus. Dealing with point ii, we could use *tree grammars* to finitely represent sets of values. This is an adequate representation for a free term algebra. Moreover, this representation permits to join two sets of constraints in a rather straightforward way (point iii) and to decide whether two value sets are disjoint (point iv). Some techniques for handling of tree grammars are described in [11].

6 A Simple Application for Security

We introduce as an example the Wide Mouthed Frog protocol, modeled in the applied π–calculus.

$$A_A = \nu n_{K_{AB}}.\overline{n_{AS}}\langle \mathsf{encrypt}(n_{K_{AB}}, n_{K_{AS}})\rangle^{l_1}.\overline{n_{AB}}\langle \mathsf{encrypt}(M, n_{K_{AB}})\rangle^{l_2}.0$$

$$A_S = n_{AS}(x_{K_{AB}}).\overline{n_{BS}}\langle \mathsf{encrypt}(\mathsf{decrypt}(x_{K_{AB}}, n_{K_{AS}}), n_{K_{BS}})\rangle^{l_3}.0$$

$$A_B = n_{BS}(x'_{K_{AB}}).n_{AB}(x_M).A_{B'}(\mathsf{decrypt}(x_M, \mathsf{decrypt}(x'_{K_{AB}}, n_{K_{BS}})))$$

$$A_{sys} = \nu n_{K_{AS}}.\nu n_{K_{BS}}.(A_A \mid A_B \mid A_S)$$

In this scenario, two participants (A and B) share a long term symmetric key with a trusted server (S): the key K_{AS} is known only to A and S, while the key K_{BS} is known only to B and S. The participant A generates a new key K_{AB} and wants to share it with B. To perform it in a secure way, A sends the new key to S, encrypting it with the key that shares with the server. The server S simply decrypts it and encrypts it again, this time using K_{BS}; then it sends it to B. Principal B receives the key from the server and an encrypted term M from A: now the continuation B' can decrypt the key from the server and then decrypt M.

The purpose of this protocol is to keep the term M secret. Hopefully, provided B' is well-behaved, an eavesdropper or an attacker can not fool the system and make it reveal M. To prove that, we model the attacker as a Dolev–Yao attacker [9]. Such an attacker can eavesdrop on every public communication channel, thus reading everything that is sent along them. In the applied π–calculus, the public channels are the unrestricted names (or, equivalently, the ones that interact with the external environment). Moreover, a Dolev–Yao attacker can intercept messages rerouting them to himself so that they don't reach the intended receiver. Finally, it can send any messages on public channels if it can build them from both data learnt by eavesdropping and public data. In the applied π–calculus, "building a message" is done by applying some function symbol in Σ to known terms.

We divide the names in secret and public ones. The names under a restriction are secret: in particular names with subscript K_{AB}, K_{AS}, K_{BS} (and their α–conversions) are secret. All the others are public. For example, those with subscript AS, BS, AB are public.

To prove the secrecy of M, we search for a CFA solution (ρ, κ) such that M can not be deduced from the union of the sets $\kappa(n)$, where n is a public name. However, note that it may be not enough to search for the minimum solution of A. This is because subject reduction ensures that the terms sent over public channels by the process A are bound by κ only *as long as* the terms sent over public channels by the attacker are bound by κ. If κ is too small, requiring the attacker's messages to be bound by κ is too strong a hypothesis and thus unfaithful to both the Dolev–Yao model and reality.

Instead, we search for a CFA solution (ρ, κ) of A such that for every public name n we have $\kappa(n) = K$ where

(i) K includes the set of public names

(ii) K is closed under function application

(iii) K does **not** contain the message M.

The first requirement models the Dolev–Yao attacker's initial knowledge, the second one models its ability to compute new messages, and the last one states the secrecy of M. Note that K represents both the values that are sent by the attacker to the process A and values that are sent by A itself; this implicitly models the fact that the attacker can intercept messages and use them to build new ones. Note that requiring the attacker to be bound by such a K is implied by the standard assumptions on the Dolev–Yao attacker. Therefore, if such a solution exists, subject reduction proves that M is kept secret even if A is under attack by a Dolev–Yao saboteur.

For instance, in the case $A_{B'} = 0$, we can find such a solution by letting $\kappa(n_{K_{AB}}) = \{n_{K_{AB}}\}^\Sigma$, $\kappa(n_{K_{AS}}) = \{n_{K_{AS}}\}^\Sigma$, $\kappa(n_{K_{BS}}) = \{n_{K_{BS}}\}^\Sigma$ and letting $\kappa(n_{AB}) = \kappa(n_{AS}) = \kappa(n_{BS}) = \rho(x_{K_{AB}}) = \rho(x'_{K_{AB}}) = \rho(x_M) = K$, where K is the minimum set of values closed under Σ–equivalence and function application that satisfies the following constraints:

$$\mathsf{encrypt}(n_{K_{AB}}, n_{K_{AS}}) \in K$$

$$\mathsf{encrypt}(M, n_{K_{AB}}) \in K$$

$$M_1 \in K \implies \mathsf{encrypt}(\mathsf{decrypt}(M_1, n_{K_{AS}}), n_{K_{BS}}) \in K$$

Since $M \notin K$, the process A preserves the secrecy of M.

7 Conclusions

We introduced a slight variant of the applied π–calculus. Many cryptographic primitive operations (encryption, decryption, signing, etc.) can be added to the applied π–calculus by modifying the equivalence relation on terms $\Sigma \vdash M_1 = M_2$. So, this calculus is particularly suited for specifying cryptographic protocols.

We defined a control flow analysis that is correct with respect to the operational semantics (subject reduction) and we discussed the existence of the minimum CFA solution and its construction.

Finally, we applied our CFA to show a simple security property.

References

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 5(46):18–36, sept 1999.

[2] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL'01), pages 104-115.*, January 2001.

[3] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols - the spi calculus. *Information and Computation*, 148, 1:1–70, January 1999.

[4] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Automatic validation of protocol narration. In *16th IEEE Computer Security Foundations Workshop (CSFW'03)*, pages 126–140. IEEE Computer Society, 2003.

[5] C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Static analysis for secrecy and non-interference in networks of processes. *Lecture Notes in Computer Science*, 2127, 2001.

[6] Chiara Bodei and Pierpaolo Degano. Static analysis for the π–calculus with applications to security. *Information and Computation*, 165:68–92, 2001.

[7] Chiara Bodei, Pierpaolo Degano, Riccardo Focardi, Roberto Gorrieri, and Fabio Martinelli. Techniques for security checking: Non-interference vs control flow analysis. In Marina Lenisa and Marino Miculan, editors, *Electronic Notes in Theoretical Computer Science*, volume 62. Elsevier Science Publishers, 2002.

[8] M. Buchholtz, H. Riis Nielson, and F. Nielson. Experiments with succinct solvers. Technical report, Informatics and Mathematical Modelling, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, Denmark, feb 2002.

[9] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, March 1983.

[10] Cedric Fournet and Martín Abadi. Hiding names: Private authentication in the applied pi calculus. In *Proceedings of the International Symposium on software Security (ISSS'02), LNCS.* Springer-Verlag, 2002.

[11] F. Nielson, H. Riis Nielson, and H. Seidl. Cryptographic analysis in cubic time. *Electronic Notes of Theoretical Computer Science*, 62, 2002.

A Alternative CFA Definitions

Speaking generally, there are two aspects that must be considered when defining a CFA:

Correctness The (static) solutions must agree with the (run–time) semantics. In our case, subject reduction must hold.

Precision CFA should have as many solutions as possible. The more solutions we have, the tighter the bounds are.

For example if we accepted as solution only the pair (ρ, κ) such that $\rho(x) = \rho(z_l) = \kappa(n) = Val$ we would be *correct* but not precise. On the other hand, allowing every pair as a solution would be very precise but not correct.

Obviously, correctness is more important than precision: we should look at correctness as a mandatory requirement and at precision as a desirable one.

We have analyzed a few variants of the CFA above; those on the conditional are the most interesting and worthy of discussion. One simple alternative to the CFA for a conditional could be

$$(\rho, \kappa) \models \begin{array}{l} \mathit{if}\ M_1 = M_2 \\ \mathit{then}\ P_1\ \mathit{else}\ P_2 \end{array} \Leftrightarrow (\rho, \kappa) \models\ P_1 \wedge (\rho, \kappa) \models\ P_2$$

However, while this is certainly correct[6], it provides us with a less precise CFA allowing fewer solutions.

The CFA for conditionals that we chose,

$$(\rho, \kappa) \models \begin{array}{l} \mathit{if}\ M_1 = M_2 \\ \mathit{then}\ P_1\ \mathit{else}\ P_2 \end{array} \Leftrightarrow \begin{array}{l} (\llbracket M_1 \rrbracket_\rho \cap \llbracket M_2 \rrbracket_\rho = \emptyset \vee (\rho, \kappa) \models\ P_1) \wedge \\ (\rho, \kappa) \models\ P_2 \end{array}$$

is more precise since it does not require the solution to be valid for the "then" branch in some cases, i.e. when it can be statically proven that the "then" branch will never be executed.

Improving precision, we could even use this variant

$$\begin{array}{l} (\rho, \kappa) \models\ \mathit{if}\ M_1 = M_2\ \mathit{then}\ P_1\ \mathit{else}\ P_2 \Leftrightarrow \\ (\llbracket M_1 \rrbracket_\rho \cap \llbracket M_2 \rrbracket_\rho = \emptyset \vee (\rho, \kappa) \models\ P_1) \wedge \\ ((\exists M : \mathsf{vars}(M) = \emptyset \wedge \llbracket M_1 \rrbracket_\rho = \llbracket M_2 \rrbracket_\rho = \llbracket M \rrbracket_\rho) \vee (\rho, \kappa) \models\ P_2) \end{array}$$

that tries to prove statically that the "else" branch is never executed. However,

[6] That is, subject reduction still holds.

this CFA is not correct as can be seen analyzing the following process:

$$!\nu n_1.\overline{n_0}\langle n_1\rangle^l.0 \mid n_0(x_0).n_0(x_1).\mathit{if}\ x_0 = x_1\ \mathit{then}\ P_1\ \mathit{else}\ P_2$$

Even if x_0 and x_1 are always bound to values that originate from "the same ν", at run–time they will have different values because n_1 must be α–converted.

Another way to refine our CFA could be to place labels on conditionals in the following way

$$\mathit{if}\ M_1 \overset{l_i,l_j}{=} M_2\ \mathit{then}\ P_1\ \mathit{else}\ P_2$$

and imposing $\rho(z_{l_i}) \supseteq [\![M_1]\!]_\rho$ and $\rho(z_{l_j}) \supseteq [\![M_2]\!]_\rho$ or (even better) some weaker constraint on ρ. This could allow some further refinement. For example in

$$\mathit{if}\ M_1 \overset{l_i,l_j}{=} M_2\ \mathit{then}\ \left(\mathit{if}\ M_1 \overset{l_k,l_m}{=} M_3\ \mathit{then}\ P_1\ \mathit{else}\ P_2\right)\ \mathit{else}\ P_3$$

it should be enough to require (i) $\rho(z_{l_k}) \supseteq \rho(z_{l_1}) \cap \rho(z_{l_j})$ instead of (ii) $\rho(z_{l_k}) \supseteq [\![M_1]\!]_\rho$. The constraint (i) could be weaker than (ii), since it could be that $\rho(z_{l_1}) \cap \rho(z_{l_j}) \subsetneq [\![M_1]\!]_\rho$. This variant appears to be both correct and more precise than our CFA. In fact our CFA abstract semantics for terms gives a *global* upper bound to run–time values of a term M, instead of a *local* one that depends on where M is used (this would require to label each occurrence of terms obtaining M^l). A similar approach can be taken on ρ: we could use *local* upper bounds for variables too.