McGill University
School of Computer Science
Sable Research Group

An Algorithmic Approach for Precise Analysis of the Π-Calculus

Sable Technical Report No. 2004-7

Sam B. Sanjabi, Clark Verbrugge
School of Computer Science, McGill University
{ssanja,clump}@cs.mcgill.ca

Abstract

We present an algorithmic approach to dataflow analysis of the π-calculus that unifies previous such analyses under a single framework. The structure of the algorithms presented lead to a dissociation of correctness proofs from the analysis itself, and this has led to the development of an algorithm that improves the accuracy of previous approaches by considering the operation of all process operators including sequencing and replication. Our analysis solution is also independent of any concurrent context in which the analyzed process runs.

1 Introduction

Recent work on language-based security has seen the application of static analysis [1] techniques applied to various formalisms in order to compute information about the possible runtime behaviour of a program prior to execution. Such analyses can then be used to formulate provable security policies that can be used as safety criteria in networking applications. Best results are naturally achieved with the most precise information flow techniques, though complex systems impose feasibility constraints. Accuracy of results, particularly with respect to relative cost of computation is thus an important quality.

In this paper we present an algorithmic approach for computing control flow analysis on the π-calculus, an extensively studied concurrent language that has been used to model and verify security protocols. Our approach leads to improvements to existing approaches by fully treating the sequentiality of potential actions in a protocol. Furthermore, we consider the potential environment that a process could concurrently be running with during our analysis, providing results that are correct independent of a concurrent context.

2 Related Work

Bodei *et al.* [2] present a static analysis of the π-calculus [3] which they later use to establish a provable information-flow property on processes [4]. The work was extended to cryptographic protocols in [5] when a static analysis on the spi-calculus [6], an extension of the π-calculus with encryption and decryption primitives, was presented. Similar techniques have been shown to be effective on Cardelli and Gordon's ambient calculus [7], where Nielson *et al.* [8] have developed an analysis that can be used for firewall validation.

Our analysis follows work done in [2], [9], [10], and [11] on the control flow analysis of the π-calculus for security purposes. In [2], Bodei et al demonstrate a CFA for the π-calculus using flow logic which computes information about the set of channel names that can be transmitted over each name in the process, as well as the set of names that a given name can *become* through input. An elementary security property regarding the leakage of information from secret to public channels was also demonstrated. In [9], we demonstrated that the analysis could be conducted via an Andersen-style points-to analysis on the `C` programming language without any label extensions to the calculus.

Previous work in this area was based on quite conservative flow and context insensitive analyses, typically only the prefixes of the process are considered, so given a process such as $x(y) + \overline{x}\langle z\rangle$, the analysis would conclude that the name z can be transmitted over the channel x, and that the name y could become z through the execution of the process. This result ignores the fact that the two

prefixes in question couldn't possibly communicate because they are on opposite sides of a choice operator.

A more accurate result could clearly be obtained by considering the structure of the process in a more sophisticated way. Inroads were made toward such a goal (again by Bodei *et al.*) in [11] in which they introduced the concept of "addressing" of subprocesses in the π-calculus. The analysis there used a slightly modified π-calculus whose semantics preserved the process structure. However, their results did not explicitly consider constructs such as replication. Furthermore, their tree representation of processes lumped sequences of prefixes into single nodes of the tree, thus losing information about the order in which they can be executed. Our analysis will include replication, and also provide results that are strictly more accurate than this approach because we consider the full sequential nature of prefixes.

3 The Π-Calculus

Our investigation of control flow is in the context of the monadic π-calculus [3], a variant of which is presented below. Assuming a countably infinite set of names $\mathcal{N} = \{a, b, \ldots, x, y, \ldots\}$, and a distinguished element τ such that $\mathcal{N} \cap \{\tau\} = \emptyset$. A *process* in the π-calculus is a term built from the following syntax:

$$\begin{array}{rcl} \pi & ::= & \tau \mid \overline{x}\langle y\rangle \mid x(y) \mid [x = y] \\ P & ::= & \mathbf{0} \mid \pi.P \mid P + P \mid P \| P \mid (\nu x)P \mid\ !P \end{array}$$

The production π defines the allowable *prefixes* (or actions) of a process P. Given a process P, the set of *output prefixes* (i.e., prefixes of the form $\overline{x}\langle y\rangle$) of P is denoted $\mathcal{O}_P$, and the set of its *input prefixes* (i.e., prefixes of the form $x(y)$) is denoted $\mathcal{I}_P$. The set of *observable prefixes* of P is defined as $\mathcal{X}_P = \mathcal{O}_P \cup \mathcal{I}_P$. Finally, the set of *match prefixes* (or tests, of the form $[x = y]$) is denoted $\mathcal{M}_P$. The name playing the role of the communication channel in an observable prefix π is called the *channel* of π (denoted $\mathrm{cha}[\pi]$), and the *datum* of π (denoted $\mathrm{dat}[\pi]$) is similarly defined. Formally, we have the following definitions:

Definition 3.1.

$$\begin{array}{lcllcl} \mathrm{cha}[x(y)] & \triangleq & x & \mathrm{cha}[\overline{x}\langle y\rangle] & \triangleq & x \\ \mathrm{dat}[x(y)] & \triangleq & y & \mathrm{dat}[\overline{x}\langle y\rangle] & \triangleq & y \end{array}$$

These functions are left undefined on other inputs.

The trailing $\mathbf{0}$ is hereafter omitted from any sequence of prefixes, i.e., π is written rather than $\pi.\mathbf{0}$ for any prefix π. Input prefixes and restrictions bind names, e.g., the name y in the process $x(y).P$ is *bound* in (or scoped to) P. Given a process P, the set of *names* $\mathrm{n}(P)$, and the set of *bound names* $\mathrm{bn}(P)$ are defined in the obvious way. The set of *free names* ($\mathrm{fn}(\cdot)$) of P is defined as $\mathrm{fn}(P) = \mathrm{n}(P) \setminus \mathrm{bn}(P)$. A process P is *closed* if $\mathrm{fn}(P) = \emptyset$, and *open* otherwise. Furthermore, the set $\beta(P)$ is defined as the set of names occurring as the bound parameter in an input prefix in P, and its complement (with respect to P) is denoted by $\chi(P) = \mathrm{n}(P) \setminus \beta(P)$.

Lastly, a notion of substitution on processes is defined as follows: given a process P, the process $P\{x \mapsto y\}$ is defined as the same process as P with every free occurrence of x in P replaced by y (under the assumption that $y \notin \mathrm{fn}(P)$). The usual conventions regarding name capture are adopted

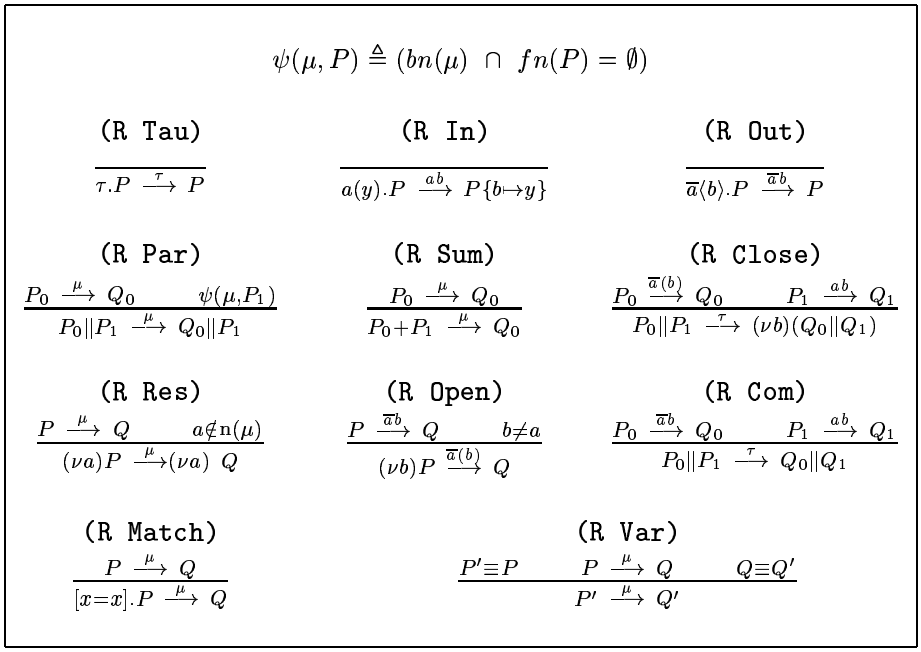

Table 1: Operational Semantics of the Π-Calculus

here as well. The α-congruence relation ($=_\alpha$) on processes is defined as usual, and allows bound names to be renamed as desired. For the rest of the paper, we assume that π-calculus processes do not repeat bound names so that analysis results for semantically distinct names are kept separate. For example, the process $(\nu a)P \| (\nu a)Q$ can be α-converted to $(\nu a)P \| (\nu a')Q\{a \mapsto a'\}$ in order to achieve this distinction.

Structural congruence ($\equiv$) on processes is also defined in the standard way. Composition and choice are associative and commutative, and $\equiv$ is transitive, reflexive and symmetric.

The complete operational semantics are given as a labelled transition system in table 1. The semantic rules define an early transition system defining a set of relations $\xrightarrow{\mu}$ between processes, with μ ranging over the set $\{ab, \overline{a}b, \overline{a}(b), \tau\}$ of *labels* for $\{a, b\} \subseteq \mathcal{N}$. A process P is said to *reduce* to a process Q if, using these inference rules, it can be proven that $P \xrightarrow{\tau} Q$. The binary predicate ψ (defined at the top of table 1) is purely a notational convenience, and is only used as a condition for the `(R Par)` rule, note that the functions n(·),bn(·), and fn(·) are extended to labels in the obvious way.

An interesting property of the π-calculus that is relevant to dataflow analysis is scope extrusion. It refers to the ability of a process to send its bound names outside of its own scope, and is a consequence of the `(R Close)` rule. Consider the parallel composition $P \| Q$ with $P \triangleq (\nu a)\overline{x}\langle a\rangle.P'$ and $Q \triangleq x(b).Q'$. The process P has a private name a, and wishes to send it on channel x, and by the `(R Open)` and `(R Out)` rules it can be derived that $P \xrightarrow{\overline{x}(a)} P'$. Meanwhile, Q is waiting to receive a value on the very same channel x, and by the `(R In)` rule, $Q \xrightarrow{xa} Q'\{b \mapsto a\}$ can be derived. Thus the `(R Close)` rule can be used to show that $P \| Q \xrightarrow{\tau} (\nu a)(P' \| Q'\{b \mapsto a\})$ is derivable. Effectively, the name a which was bound in P has been sent to another process, and its scope has been "extruded" to encompass both processes. This phenomenon is obviously important when trying to develop dataflow analyses that are correct independent of context.

4 General Approach

In the literature [4, 11], the solution of a dataflow analysis of a π-calculus process P is often expressed as a pair of functions (R, K). The function $R : \mathcal{N} \to 2^{\mathcal{N}}$ assigns to each name in P a set of names which it could possibly assume during execution. This set is non-trivial for names that are bound by an input prefix because they are the targets of substitution during a reduction sequence. The function $K : \mathcal{N} \to 2^{\mathcal{N}}$ assigns to each name in P a set of names which could be transmitted over the input name. This set is non-trivial for names that are not bound by input in P. Denote the set of all such pairs of functions by $\mathcal{L}_\emptyset$.

Rather than following a flow logic approach as was previously done, we follow an algorithmic approach that clarifies the inner workings of current algorithms. A control flow analysis is fully defined by the following components:

1. A notion of an approximate representation of the process being analyzed
2. An iterative function on this representation that refines the current solution at each step.

A very basic analysis of a π-calculus process P may represent P by its set of communicable prefixes $\mathcal{X}_P$, call this representation $[\![P]\!]_\emptyset$. An analysis function $\Psi_{[\![P]\!]_\emptyset} : \mathcal{L}_\emptyset \to \mathcal{L}_\emptyset$ can be defined as follows:

1. For each input prefix $\pi_i \in [\![P]\!]_\emptyset$ augment $R(\mathrm{dat}[\pi_i])$ by adding the following set:

$$\bigcup_{x \in R(\mathrm{dat}[\pi_i])} K(x)$$

2. For each output prefix $\pi_o \in [\![P]\!]_\emptyset$ augment $K(x)$ such that $x \in R(\mathrm{cha}[\pi_o])$ by adding $R(\mathrm{dat}[\pi_o])$

Clearly, iteration of this process will eventually reach a fixed point as there are a finite number of names in P, and no names are ever removed from the image sets of R and K. In fact, this flow analysis of the calculus has a strong connection with Andersen's points-to analysis on the C programming language (see our work in [9]), in which a program is represented by its set of pointer assignment statements.

A slightly more sophisticated analysis would use a representation $[\![P]\!]_\mathcal{M}$ that pairs each prefix with the set of match prefixes that precede it. The above update rules are then only applied for a given prefix if it has been shown that all the matches that precede it could potentially pass. This can be done by first testing the truth of the predicate $R(x) \cap R(y) \neq \emptyset$. It is not difficult to see that such an analysis is identical to the flow analysis provided by Bodei *et al.* in [4].

Generally, we can define a control flow analysis on the π-calculus $\mathcal{A}_\pi$ by a tuple $\mathcal{A}_\pi = \langle \mathcal{L}, [\![\cdot]\!], f : \mathcal{L} \to \mathcal{L}, U \rangle$ where:

- $\mathcal{L}$ is the space of solutions
- $[\![\cdot]\!]$ provides an abstract representation of a process
- $f : \mathcal{L} \to \mathcal{L}$ is an iterative function that can be viewed as a set of update rules as above.
- U is a unary predicate that dictates when the update rules are to be applied

The following sections provide a new control flow analysis algorithm by defining further refinements of each of these components. We begin by expressing the basic definitions required to understand the rest of the paper.

5 Basic Definitions

Given any countable set S, we define $S_\perp$ as the complete lattice comprising of the elements of S with two elements $\perp$ and $\top$ such that $\forall s \in S : \perp \sqsubseteq s \sqsubseteq \top$, and the elements of S are incomparable. The following standard lattice-theoretic facts are used in our analysis:

Fact 5.1. *Given any countable set S, the set of all subsets of S, denoted 2^S, forms a complete lattice with partial ordering $\subseteq$, bottom and top elements $\emptyset$ and S, and $\cup$ and $\cap$ as the join and meet operations respectively.*

Fact 5.2. *Complete lattices are closed under Cartesian product, i.e., given lattices $\mathcal{L}_1$ and $\mathcal{L}_2$, then $\mathcal{L}_1 \times \mathcal{L}_2$ is also a complete lattice with all operations defined point-wise.*

Fact 5.3. *Given a countable set S and a complete lattice $\mathcal{L}$, the set of all total functions from S to $\mathcal{L}$ (denoted $[S \Rightarrow \mathcal{L}]$) is also a complete lattice with all operations defined point-wise.*

We formalize prefixes by defining $\mathcal{O} = \langle \mathcal{N} \times \mathcal{N} \rangle$ and $\mathcal{I} = (\mathcal{N} \times \mathcal{N})$ as the sets of output and input prefixes respectively, and $\mathcal{X} = \mathcal{O} \cup \mathcal{I}$ as the set of all non silent prefixes. Using these set-theoretic definitions, the functions $\mathrm{cha}[] : \mathcal{X} \to \mathcal{N}$ and $\mathrm{dat}[] : \mathcal{X} \to \mathcal{N}$, can be viewed as the appropriate projections over these sets. We further define the set $\mathcal{M} = [\mathcal{N} \times \mathcal{N}]$ as the set of all match prefixes. The bracketing around the cartesian products are intended to indicate that these sets are isomorphic but kept distinct, thus an output prefix $\overline{x}\langle y \rangle \in \mathcal{O}$ is writen as $\langle x, y \rangle$ and so on.

Given a set of observable prefixes $S \subseteq \mathcal{X}$, the partitions of input and output prefixes of S are denoted $S \restriction \mathcal{I} = S \cap \mathcal{I}$ and $S \restriction \mathcal{O} = S \cap \mathcal{O}$ respectively. The $\restriction$ operator can be read as "projected onto".

Given a function $f : X \to Y$, and given $x \in X$ and $y \in Y$, we use the standard notion of substitution on functions by defining $f\{x \mapsto y\}$ to be the same function as f, except that $(f\{x \mapsto y\})(x) = y$.

6 Solution Space

A constructive flow analysis algorithm can be expressed as a monotone function $f : \mathcal{L} \to \mathcal{L}$ operating over a complete lattice of solutions $\mathcal{L}$. By elementary fixed point theory, a least solution can be computed by iterating f from bottom in the lattice. This section describes the solution space of our algorithm and proves that it is a complete lattice, while the following sections describe the process representation, the update rules, and the update condition respectively.

In order to develop a concurrent context independent analysis, the traditional solution space must be expanded by a third element [11] to track the knowledge of the environment. This knowledge is represented by a set $E \subseteq \mathcal{N}$, expanding the solution space to a triple (R, K, E). This step was taken in [11] to give a treatment of process environments.

Furthermore, for an accurate analysis that considers the sequential order of possible actions, we need to keep track of which prefixes communicate at each step. One way to do this is by carrying

a function $\mathcal{C} : \mathcal{X} \to \mathbb{B}$, for $\mathbb{B}$ the two point boolean lattice $\mathbb{ff} \sqsubseteq \mathbb{tt}$, leaving the final solution space as the 4-tuple $(R, K, E, \mathcal{C})$. In order to distinguish between multiple occurrences of the same prefix, we actually require that each prefix in a process P be labelled with elements of a countable label set, and that $\mathcal{C}$ actually take these labels as inputs. However, we make the clarifying assumption that multiple occurrences of the same prefix do not occur in P, as adding labelling would only clutter the syntax and is a rather trivial extension.

The complete solution space $(R, K, E, \mathcal{C})$ forms a complete lattice because each of its components forms a complete lattice:

Proposition 6.1. *The following sets are complete lattices:*

1. $\mathcal{L}_R = \mathcal{L}_K = [\mathcal{N} \Rightarrow 2^{\mathcal{N}}]$
2. $\mathcal{L}_E = 2^{\mathcal{N}}$
3. $\mathcal{L}_{\mathcal{C}} = [\mathcal{X} \Rightarrow \mathbb{B}]$

Proof. Trivial from facts 5.1 and 5.3. □

From fact 5.2, the set $\mathcal{L}_R \times \mathcal{L}_K \times \mathcal{L}_E \times \mathcal{L}_{\mathcal{C}}$ (i.e., the set of all potential solutions $(R, K, E, \mathcal{C})$) is thus also a complete lattice.

6.1 Initialization

Defined as above, the bottom element of our lattice has the functional components (namely R, K, and $\mathcal{C}$) returning bottom in their respective image lattices for any input, and the bottom of the E component $\emptyset$. However, this is not enough information to reflect the initial information known about a process before analysis begins. For instance, the initial environment knowledge is not empty, but in fact consists of all the free names of P *as well as* any names not in P. Thus, in order for our analysis to compute meaningful information, we must iterate our analysis function from a solution properly reflecting the initial state, which is dependent on the process being analyzed. We must now show that this initial state preserves the lattice properties required for the function to terminate.

The following information is known about the solution prior to the execution of the process P:

- Every name in P not bound by input always represent themselves (as they can never be the target of a substitution). Formally, the condition $\forall x \in \chi(P) : R(x) = \{x\}$ must hold initially.
- The environment knows about all of the free names of P and all of the names not in P.

The second item is handled by introducing a special name Φ_P which represents the set $\{x \in \mathcal{N} \mid x \notin \mathrm{n}(P)\}$. Inclusion in Φ_P is thus tested by non-inclusion in $\mathrm{n}(P)$ for any process P, and the environment knowledge component E of the solution is thus initialized to the set $\mathrm{fn}(P) \cup \{\Phi_P\}$. We now show that the desirable properties of our solution space still hold:

Proposition 6.2. *Given a countable set S, its induced function space lattice $\mathcal{L}_S = [S \Rightarrow 2^S]$, and given any subset $I \subseteq S$, define the function $\bot_S^I$ as*

$$\bot_S^I(x) = \begin{cases} \{x\} & , \text{ if } x \in I \\ \emptyset & , \text{ otherwise} \end{cases}$$

for any $x \in S$. Then the subset $\mathcal{L}_S^I = \{f \in \mathcal{L}_S \mid \bot_S^I \sqsubseteq f\}$ forms a complete sub-lattice of $\mathcal{L}_S$ with the same partial ordering, least upper bound operation, greatest lower bound operation, top element, and with bottom element $\bot_S^I$.

Proof. The relation $\sqsubseteq$ is trivially still a partial ordering on the subset $\mathcal{L}_S^I$, and $\bot_S^I \sqsubseteq f$ for any $f \in \mathcal{L}_S^I$ by definition, and thus in fact is the bottom element of $\mathcal{L}_S^I$. Since $\top_S \sqsupseteq f$ for any $f \in [S \Rightarrow 2^S]$ then it is certainly greater than any function in $\mathcal{L}_S^I$, so the top element does not change. For any subset $F \subseteq \mathcal{L}_S^I$, $\bot_S^I$ is a lower bound of F by definition of $\mathcal{L}_S^I$. Furthermore, $\sqcup F$ is an upper bound of F by definition of $\sqcup$. Thus $\bot_S^I \sqsubseteq \sqcup F$ by transitivity of $\sqsubseteq$, and hence $\sqcup F \in \mathcal{L}_S^I$ by the definition of $\mathcal{L}_S^I$. By the duality of $\sqcup$ and $\sqcap$, $\sqcap F \in \mathcal{L}_S^I$ yielding the result. □

Proposition 6.3. *Given a countable set S, its power set lattice $\mathcal{L}_S = 2^S$, and given any subset $\bot_S^I = I \subseteq S$, then the subset $\mathcal{L}_S^I = \{T \in \mathcal{L}_S \mid \bot_S^I \sqsubseteq T\}$ forms a complete sub-lattice of $\mathcal{L}_S$ with the same partial ordering, least upper bound and greatest lower bound operations, and top element, and with bottom element $\bot_S^I$.*

Proof. Since every subset in $\mathcal{L}_S^I$ includes $\bot_S^I$, the result follows trivially. □

These two propositions allow us to restrict our solution space by taking the following sub-lattices of $\mathcal{L}_R$ and $\mathcal{L}_E$:

- $\mathcal{L}_R$ is restricted to the sub-lattice $\mathcal{L}_R^{\chi(P)}$ according to the procedure in proposition 6.2.
- $\mathcal{L}_E$ is restricted to the sub-lattice $\mathcal{L}_E^{\text{fn}(P) \cup \{\Phi_P\}}$ according to the procedure in proposition 6.3.

Thus, for any process P, our initialized solution space lattice $\mathcal{L}_\Psi^P$ is defined as the following:

$$\mathcal{L}_\Psi = \mathcal{L}_R^{\chi(P)} \times \mathcal{L}_K \times \mathcal{L}_E^{\text{fn}(P) \cup \{\Phi_P\}} \times \mathcal{L}_\mathcal{C}$$

with bottom element $\bot_\Psi^P = (\bot_R^{\chi(P)}, \bot_K, \bot_E^{\text{fn}(P) \cup \{\Phi_P\}}, \bot_\mathcal{C})$. It follows from the results above that $\mathcal{L}_\Psi^P$ forms a complete lattice for any process P, and thus iterating a monotone function from $\bot_\Psi^P$ will also reach a fixed point. The superscript P is heretofore omitted when the process in question is clear from the context.

7 Process Representation

Having characterized the solution space $\mathcal{L}_\Psi$, we now define an appropriate representation of processes that contains enough information about the process structure to yield an accurate solution.

In order to track dataflow in the process accurately, we must define which prefixes can potentially communicate with one another. In a flow-insensitive analysis such as [4], where composition and

choice operators are not taken into account, it is assumed that every input prefix can communicate with every output prefix and vice versa. The enhanced analysis of Bodei *et al.* in [11] accounts for the fact that prefixes on the opposite sides of a choice operator cannot communicate, but doesn't account for the fact that a prefix cannot communicate until all of its predecessors have had a chance to do so. Thus we develop a representation of processes that allows us to reason about the latter.

The first step towards such a representation is the generation of the parse tree of a process P. We explicitly define this here in order to illustrate how the sequencing of prefixes is taken into account. Our definition of a parse tree consists of a set of nodes defined by the following grammar:

$$\begin{aligned} \text{Node} &::= \text{PrefixNode} \mid \text{OpNode} \mid \text{PrefixNodeSet} \\ \text{PrefixNode} &::= (\pi_{\text{this}}, \pi_{\text{pred}}) \\ \text{PrefixNodeSet} &::= \{\text{PrefixNode}_1, \ldots, \text{PrefixNode}_n\} \\ \text{OpNode} &::= \| \mid + \end{aligned}$$

A node is either a pair of prefixes (the first being the prefix at the current position in the process, and the second being a pointer to the preceding prefix), a process operator, or a set of prefix pairs (used to represent a replicated process, where flow information is lost due to the nature of the replication operator). We define projections **this**[PrefixNode] and **pred**[PrefixNode] providing access to the elements of a PrefixNode. Recalling that the projection operator $\restriction$ is used to partition a set of prefixes into it's input and output subsets, we extend it to PrefixNodes in the obvious way:

Definition 7.1. Given a PrefixNodeSet S,

$$\begin{aligned} S \restriction \mathcal{O} &= \{n \in S \mid \mathbf{this}[n] \in \mathcal{O}\} \\ S \restriction \mathcal{I} &= \{n \in S \mid \mathbf{this}[n] \in \mathcal{I}\} \end{aligned}$$

Process trees are then defined as follows:

$$\begin{aligned} \text{ProcTree} &::= \varepsilon \mid \langle \text{Node}, \text{ProcTreeSet} \rangle \\ \text{ProcTreeSet} &::= \{\text{ProcTree}_1, \ldots, \text{ProcTree}_n\} \end{aligned}$$

A tree is either an empty tree ε, or the pairing of a Node (accessed by the projection **root**[ProcTree]) and a set of ProcTrees. The function $\mathbf{child}_i$[ProcTree] returns the i^{th} ProcTree in the set. A PrefixNode N is defined to be in a ProcTree T (denoted $N \in T$) if N is a node of T, or $N \in S$ where S is a PrefixNodeSet that is a node of T.

Given these definitions, we define the abstract representation of a process P by a function $[\![P]\!]_T^C$ taking a π-calculus process P and returning a ProcTree (the element C is used to keep track of the last prefix seen). First, we define a flow-insensitive representation $[\![P]\!]_S^C$ that generates a PrefixNodeSet, as this will be our representation of a replicated process:

Definition 7.2.

$$\begin{aligned} [\![\mathbf{0}]\!]_S^C &= \emptyset \\ [\![P \| Q]\!]_S^C &= [\![P]\!]_S^C \cup [\![Q]\!]_S^C \\ [\![P + Q]\!]_S^C &= [\![P]\!]_S^C \cup [\![Q]\!]_S^C \\ [\![(\nu x)P]\!]_S^C = [\![\tau.P]\!]_S^C = [\![!P]\!]_S^C &= [\![P]\!]_S^C \\ [\![\pi.P]\!]_S^C &= \{(\pi, C)\} \cup [\![P]\!]_S^\pi \end{aligned}$$

This representation generates a set of prefixes, each paired with its sequential predecessor. The tree representation can now be defined inductively on the structure of P

Definition 7.3.

$$\begin{aligned}
[\![\mathbf{0}]\!]_T^C &= \varepsilon \\
[\![P\|Q]\!]_T^C &= \langle \|, \{[\![P]\!]_T^C, [\![Q]\!]_T^C\}\rangle \\
[\![P+Q]\!]_T^C &= \langle +, \{[\![P]\!]_T^C, [\![Q]\!]_T^C\}\rangle \\
[\![(\nu x)P]\!]_T^C = [\![\tau.P]\!]_T^C &= [\![P]\!]_T^C \\
[\![\pi.P]\!]_T^C &= \langle (\pi, C), \{[\![P]\!]_T^\pi\}\rangle \\
[\![!P]\!]_T^C &= [\![P]\!]_S^C
\end{aligned}$$

Notice that $[\![\cdot]\!]_T^C$ generates a ProcTree in which every node has no more than two children. However, since our structural congruence is associative, we identify the trees $[\![(P\|Q)\|R]\!]_T = [\![P\|(Q\|R)]\!]_T$ using the following transformation (the subscript C is omitted for clarity):

$$(\|, \{[\![P\|Q]\!]_T, [\![R]\!]_T\}) = (\|, \{[\![P]\!]_T, [\![Q\|R]\!]_T\}) \rightsquigarrow (\|, \{[\![P]\!]_T, [\![Q]\!]_T, [\![R]\!]_T\})$$

Repeatedly applying this transformation to closure clearly terminates on a finite process, and simply collects all of the subprocesses P_i in a process $P_1\| \cdots \|P_n$ under a common OpNode. A similar transformation is performed for collections of processes that are choiced together.

The final tree representation $[\![P]\!]_\Psi$ of a process P is computed as $[\![P]\!]_T^\emptyset \rightsquigarrow^* [\![P]\!]_\Psi$, where $\emptyset$ indicates that there is no preceding prefix at the top level of the process. This representation simply generates the parse tree of P, with the exception of a replicated process $!P$, which is interpreted as the set of its prefixes. This is because any prefix in a replicated process can potentially communicate with any other prefix because of the structural congruence rule $!P \equiv P\|!P$.

We complete the definition of the representation of P by defining the function $\mathcal{E}$: ProcTree $\rightarrow$ (PrefixNode $\times$ PrefixNode) generating the pairs of prefixes that can *structurally* communicate:

Definition 7.4. Given a π-calculus process P, define the set of structurally communication enabled prefixes as $\mathcal{E}([\![P]\!]_\Psi)$, with $\mathcal{E}$ the following function:

$$\begin{aligned}
\mathcal{E}(\varepsilon) &= \emptyset \\
\mathcal{E}(\langle \text{PrefixNode}, \{T\}\rangle) &= \mathcal{E}(T) \\
\mathcal{E}(\langle +, \{T_1, \ldots, T_n\}\rangle) &= \textstyle\bigcup_i \mathcal{E}(T_i) \\
\mathcal{E}(\langle \|, \{T_1, \ldots, T_n\}\rangle) &= \Delta(\{T_1, \ldots, T_n\}) \cup \textstyle\bigcup_i \mathcal{E}(T_i) \\
\mathcal{E}(\langle S, \emptyset\rangle\rangle) &= (S \upharpoonright \mathcal{O}) \times (S \upharpoonright \mathcal{I})
\end{aligned}$$

where T,T_1,...,T_n are ProcTrees and S is a PrefixNodeSet.

The first equation handles the empty tree. The next says that a PrefixNode does not generate any new pairs. The third handles the case that prefixes on opposite sides of a choice operator can't communicate by returning all the pairs in the summed children without generating any pairs *between* the prefixes in them. The fourth equation handles processes in parallel composition by generating the same set of pairs within each child tree *in addition to* the set of pairs $\Delta(\{T_1, \ldots, T_n\})$, where this set comprises each input (output) PrefixNode in T_i paired with each output (input) PrefixNode in every $T_j \neq T_i$ for all i. The final equation handles the case of a replicated sub-process by generating the set of all input/output pairs in the given NodeSet S.

This structure differs from the structure in [11] in that each prefix in a non-replicated process resides in its own node and knows who its predecessor is. We can now proceed to define a set of update rules, and an update condition that take advantage of this structure.

8 The Analysis Function (Update Rules)

Our update function $f_\Psi : \mathcal{L}_\Psi \to \mathcal{L}_\Psi$ will iterate over each prefix pairing $\eta \in [\![P]\!]_\Psi$, applying the following rules to each component of the tuple $(R, K, E, \mathcal{C}) \in \mathcal{L}_\Psi$:

1. for each η such that $\pi_o = \mathbf{this}[\eta] \in \mathcal{O}_P$, do the following:
   - augment each $K(x)$ such that $x \in R(\mathrm{cha}[\pi_o])$ by adding $R(\mathrm{dat}[\pi_o])$
   - if $R(\mathrm{cha}[\pi_o]) \cap E \neq \emptyset$, then add $R(\mathrm{dat}[\pi_o])$ to E
2. for each η such that $\pi_i = \mathbf{this}[\eta] \in \mathcal{I}_P$, do the following:
   - augment $R(\mathrm{dat}[\pi_i])$ by adding
   $$\bigcup_{x \in R(\mathrm{cha}[\pi_i])} K(x)$$
   - if $R(\mathrm{cha}[\pi_i]) \cap E \neq \emptyset$, then also add E
3. Set $\mathcal{C}(\mathbf{this}[\eta])$ to $\mathbb{t}$.

Rules (1) and (2) are identical to the rules described in [11], with the exception that the environment now includes the special name Φ_P. Intuitively, these rules encode all possible communications into the solution with respect to the input pair (R, K). The last rule indicates that the prefix in question has been shown to communicate. Intriguingly, the actual analysis function does not change much from previous analyses. We shall see that most of the machinery of the analysis lies in the condition under which this function is applied.

9 The Update Condition

Given the tree $T = [\![P]\!]_\Psi$ representing a process P, we wish to define an update condition U_Ψ with the following features:

1. U_Ψ should not consider the communication contribution of a prefix if that prefix is blocked by a preceding prefix that is unable to communicate
2. U_Ψ should produce a result that is correct independent of the context in which P runs

Regarding point (2) we follow the approach of [11] by asserting that a prefix can communicate with the environment if the the latter has knowledge of its channel. Formally, we have the following definition:

Definition 9.1. Given a tuple $\lambda = (R, K, E, \mathcal{C}) \in \mathcal{L}_\Psi$, and a ProcTree T, then a PrefixNode $\eta \in T$, where $\pi = \mathbf{this}[\eta] \in \mathcal{X}$, is *environment enabled*,denoted $\lambda \vdash_E \eta$, if and only if

$$E \cap R(\mathrm{cha}[\pi]) \neq \emptyset$$

Recalling that cha[·] denotes the name acting as the channel in the given prefix.

The notable difference between this definition and that of [11] is the addition of the name Φ_P to the knowledge of the environment.

Similarly, a condition for a match prefix to be able to pass can be encoded as follows:

Definition 9.2. Given a tuple $\lambda = (R, K, E, \mathcal{C}) \in \mathcal{L}_\Psi$, and a ProcTree T, then a PrefixNode $\eta \in T$, where $[x, y] = \mathbf{this}[\eta] \in \mathcal{M}$ is *match enabled*, denoted $\lambda \vdash_M \eta$, if and only if

$$R(x) \cap R(y) \neq \emptyset$$

The other condition of a prefix being able to communicate is if there exists another prefix in the process that it can communicate with:

Definition 9.3. Given a tuple $\lambda = (K, R, E, \mathcal{C})$, and a ProcTree T, an input/output PrefixNode pair $(\eta_i, \eta_o) \in \mathcal{E}(T)$, where $\pi_i = \mathbf{this}[\eta_i] \in \mathcal{I}$ and $\pi_o = \mathbf{this}[\eta_o] \in \mathcal{O}$ are *communication enabled*, denoted $\lambda \vdash_C (\eta_i, \eta_o)$, if and only if:

$$R(\mathrm{cha}[\pi_i]) \cap R(\mathrm{cha}[\pi_o]) \neq \emptyset$$

This states that an input/output PrefixNode pair is enabled if the set of names that their channels could be (given by R) overlap. This precisely captures the notion of two prefixes being able to communicate in isolation. The handling of the sequential nature of prefixes in point (1) can thus be formally encapsulated by the following definition:

Definition 9.4. Given a tuple $\lambda = (K, R, E, \mathcal{C})$, and a ProcTree T, a PrefixNode $\eta \in T$ is *enabled* (without qualification), denoted $\lambda \vdash \eta$, if and only if:

$$\mathcal{C}[\mathbf{pred}[\eta]] \bigwedge \begin{cases} \lambda \vdash_M \eta & \text{if } \mathbf{this}[\eta] \in \mathcal{M} \\ \lambda \vdash_E \eta \quad \vee \quad \Big(\exists \eta' \in T : \quad \lambda' \vdash_C (\eta, \eta') \wedge & \\ \qquad\qquad \mathcal{C}[\mathbf{pred}[\eta']]\Big) & \text{if } \mathbf{this}[\eta] \in \mathcal{X} \end{cases}$$

This definition simply says that a PrefixNode can communicate in the context $(K, R, E, \mathcal{C})$ if it's predecessor can communicate (i.e., it's not blocked by preceding prefixes), and it is either match enabled (if it is a match), environment enabled, or there is an unblocked prefix with which it is communication enabled. This then becomes the definition of our update condition U_Ψ:

Definition 9.5. Given any π-calculus process P, any tuple $\lambda \in \mathcal{L}_\Psi$, and any prefix pairing $\eta \in [\![P]\!]_\Psi$ define the unary predicate U_Ψ by

$$U_\Psi(\eta) \Leftrightarrow \lambda \vdash \eta$$

Thus, in each iteration, the updates given by the definition of f_Ψ in the previous section are only applied if the η under consideration satisfies $U_\Psi(\eta)$. Our full control flow analysis is then given by the tuple

$$\mathcal{A}_\pi^\Psi(P) = \langle \mathcal{L}_\Psi, [\![P]\!]_\Psi, f_\Psi, U_\Psi \rangle$$

It should be fairly simple to check that previous flow analysis approaches ([11], [4]) would compute that the process $P \equiv (\nu a)\overline{a}\langle b \rangle.(\overline{c}\langle d \rangle \| c(e))$ has a solution strictly greater than $\bot_\Psi$, whereas $\mathcal{A}_\pi^\Psi(P) = \bot_\Psi$ (correctly, because the process is unable to reduce in any context). Thus the improved accuracy of our algorithm is evident. The proof of the correctness of the approach is too long to be included here in full (but it can be found in [12]). A sketch is provided in the next section.

10 Sketch of Correctness Proof

The proof of correctness of our algorithm proceeds in two stages:

1. Proving the correctness of the algorithm under the assumption that the update condition U_Ψ always passes
2. Proving that no actual communications are excluded by U_Ψ when it is applied as defined above

The proof of (1) can easily be done with a slight modification of the flow logic in [11] to apply to the calculus presented in this paper. However, it was proved a different way in [12]. In the latter, each component of the analysis solution $\mathcal{A}^\Psi_\pi(P) = (R, K, E, \mathcal{C})$ is compared to the components of a tuple $\lambda = (\widetilde{R}, \widetilde{K}, \widetilde{E}, \widetilde{\mathcal{C}})$ in $\mathcal{L}_\Psi$ which is generated from the actual traces of a π-calculus process. For the R component, this was done by defining a function Λ_R taking a process P to a function of the form R as follows:

- Observe that the only actions in a trace $P \xrightarrow{\tau} P_0 \xrightarrow{\tau} \ldots$ that affect dataflow are communications. Thus, from this trace, a sequence of such "communication actions" (expressed as pairs of prefix *occurences*) is generated.
- Now observe that our flow analysis identifies the action pairs where both prefix occurences are from a replicated sub-process. Thus these "repeated communications" are removed from the action sequence. It is shown in [12] that this yields a finite sequence of actions.
- For each prefix in such a sequence, there exists a finite number (in fact, at most 3) of preceding actions that may have led to a name substitution in the prefixes of the action. Since the sequence is finite, setting these preceding actions as the "children" of the current action yields a structure we call the "dependency tree" of the action.

The function Λ_R is then defined by updating the R set for each action in every trace of the process, i.e., if the prefix pair $(a(b), \overline{a}\langle c\rangle)$ is in some trace of P, then $c \in \Lambda_R(P)(b)$. The following lemma can then be proved by reasoning inductively on the dependency tree of an arbitrary action:

Lemma 10.1. *Given any π-calculus processes P and C, and a flow analysis $\mathcal{A}'_\pi = \langle \mathcal{L}_\Psi, [\![\cdot]\!]_\Psi, f_\Psi, \mathbf{1}\rangle$ such that $(R, K, E, \mathcal{C}) = \mathcal{A}'_\pi(P)$, then*

$$\Lambda_R(C \| P) \nabla P \sqsubseteq R$$

Where the ∇ operator is defined by the restriction of the domain of the R-function given as its left operand to the names appearing in the process given as its right operand. Note that this theorem illustrates that the solution computed for a process P is independent of any context C that may be running in parallel with it. Similar theorems can be proved for the other components of the solution, but this latter property is lost. For instance, for the K component, it is impossible to ask for this condition without trivializing the solution because the context C may very well transmit data over a channel name that appears in P but is never used by it.

The second part of the proof is to show that the condition U_Ψ does not exclude any communication that occurs in the traces of $C \| P$. This is done by defining an operator $\Uparrow$ on processes by $P \Uparrow S$

denoting the same process as P truncated at all the prefix occurences η in the set S where $U_\Psi(\eta)$ remains false at the end of the analysis. The second part of the theorem is then proved by the following lemma:

Lemma 10.2. *Given π-calculus processes P and C, and the usual flow analysis $\mathcal{A}_\pi^\Psi$, then*

$$\Lambda_R(C\|P) = \Lambda_R((C\|P) \Uparrow S)$$

where $S = \{\eta \in [\![P]\!]_\Psi \mid U_\Psi(\eta) = \mathit{ff}$ after the analysis$\}$.

11 Future Work and Conclusions

We have defined an algorithmic framework in which flow analyses of the π-calculus can be defined and compared, and have shown how it can be used to define more accurate analyses than the existing state of the art. However, there are still many issues to be resolved. First, there are further approximation points that can be refined in order to provide more accurate results. For instance, Colussi *et al.* [13] have recently shown that greater accuracy can be obtained by considering sequences of match prefixes together rather individually. Another possible avenue for improvement would be the exploration of a better treatment of the replication prefix, which is still being abstracted rather naively. Such work would involve improvements to the representation and update condition of the analysis presented.

Our analysis, while accurate, suffers from a lack of efficiency because the primary focus of this work was the creation of a unified model in which analyses of the π-calculus (and potentially other languages) can be defined. However, while our approach immediately leads to an algorithm in all cases, such an implementation may not be the fastest way to compute the function. For instance, the analysis of [4] which runs in $O(N^5)$ time, has the same complexity when defined in the framework here. However, it has been shown in [10] that this function can be computed in cubic time by using the logical concept of Horn clauses with sharing. Thus it is left for future work to determine if the precise analysis presented here can be shown to run more efficiently.

References

[1] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag (1999)

[2] Bodei, C., Degano, P., Nielson, F., Nielson, H.R.: Control flow analysis for the π-calculus. In: Proceedings of CONCUR '98. Volume 1466 of Lecture Notes In Computer Science., Springer-Verlag (1998) 84–98

[3] Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, parts I and II. Information and Computation **100** (1992) 1–77

[4] Bodei, C., Degano, P., Nielson, F., Nielson, H.R.: Static analysis for the π-calculus with applications to security. INFCTRL: Information and Computation (formerly Information and Control **168** (2001)

[5] Bodei, C., Degano, P., Nielson, F., Nielson, H.R.: Static analysis for secrecy and non-interference in networks of processes. In: Proceedings of the 6^{th} International Conference on Parallel Computing Technologies. (2001) 27–41

[6] Abadi, M., Gordon, A.: A calculus for cryptographic protocols - the spi-calculus. Information and Computation **148** (1999) 1–70

[7] Cardelli, L., Gordon, A.D.: Mobile ambients. In: Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98, Springer-Verlag, Berlin Germany (1998)

[8] Nielson, F., Nielson, H.R., Hansen, R.R., Jensen, J.G.: Validating firewalls in mobile ambients. In: International Conference on Concurrency Theory. (1999) 463–477

[9] Sanjabi, S.B., Verbrugge, C.: Points-to inspired static analysis for the π-calculus. Sable Technical Report 2003-02, McGill University, School Of Computer Science (2003)

[10] Nielson, F., Seidl, H.: Control-flow analysis in cubic time. In: Proc. ESOP '01. Number 2028 in Lecture Notes in Computer Science, Springer-Verlag (2001) 252–268

[11] Bodei, C., Degano, P., Priami, C., Zannone, N.: An enhanced CFA for security policies. In: Proceedings of the Workshop on Issues on the Theory of Security (WITS '03), Warszawa (2003) 131–145

[12] Sanjabi, S.: Dataflow analysis of the π-calculus. Master's thesis, McGill University, School of Computer Science (2004)

[13] Colussi, L., Filè, G., Griggio, A.: Precise analysis of π-calculus in cubic time. In: Proceedings of the 3^{rd} IFIP International Conference on Theoretical Computer Science. (2004)