McGill University
School of Computer Science
Sable Research Group

Precise, Partially-Compositional Analysis of the π-Calculus

Sable Technical Report No. 2004-01

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

Abstract

We present a new algorithm for computing control flow analysis on the π-calculus. Our approach is strictly more accurate then existing algorithms, while maintaining a polynomial running time. Our algorithm is also partially compositional, in the sense that the representational structure that results from analyzing a process can be efficiently reused in an analysis of the same process in another context.

1 Introduction

Recent work on language-based security has seen the application of static analysis [8] 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 a new algorithm for computing control flow analysis on the π-calculus. Our approach is strictly more accurate then existing algorithms, while maintaining a polynomial running time.

Our algorithm is also partially compositional, in the sense that the representational structure that results from analyzing a process can be efficiently reused in an analysis of the same process in another context. Compositionality is an important practical consideration when the process or protocol under analysis is best understood or needs to be verified in various contexts. By reusing all or some of the information already computed about a subprocess the overall computation effort can be reduced.

The main contributions of this paper consist of:

- A new polynomial time algorithm for control flow analysis of the π-calculus with strictly greater accuracy than the current state of the art. Our focus is initially on the calculus without replication, but a simple extension to include replication is provided.
- A partially compositional approach to control flow analysis. The representation structure we use during computation of our algorithm is independent of context, and so can be reused when analyzing the same subprocess in another context. This is a partial step toward an accurate, efficient and fully compositional analysis.

We also indicate some directions for improving algorithm efficiency by tightening conservative estimates of limits on process syntactic entities, and reducing redundancy in the constraint solving process.

1.1 Roadmap

Section 2 describes related approaches, and section 3 gives details on the context of our approach. Basic definitions for our algorithm are given in section 4, and an initial version of the algorithm is

developed in section 5. The algorithm is then refined into a more accurate, incremental approach in section 6. Sections 7 and 8 discuss the algorithm and possible extensions and conclude.

2 Related Work

Bodei et al [4] present a static analysis of the π-calculus [7] which they later use to establish a provable information-flow property on processes [3]. The work was extended to cryptographic protocols in [1] when a static analysis on the spi-calculus [6], a concurrent language with encryption and decryption primitives, was presented. Similar techniques have been shown to be effective on Cardelli and Gordon's ambient calculus [2], where Nielson et al [11] have developed an analysis that can be used for firewall validation, and more recently presented an abstract interpretation of mobile ambients [10] used for syntactic checks. An abstract interpretation based on game semantics has also been developed by Malacaria and Hankin [12] with which they compute a notion of information flow on a sequential program [13].

Our analysis follows work done in [4], [14], [9], and [5] on the control flow analysis of the π-calculus for security purposes. In [4], 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 [14], 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 $x(y) + \bar{x}z$, 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 [5] in which they introduced the concept of "addressing" of a π-calculus process. The analysis there used a slightly modified (but equally expressive) π-calculus whose semantics preserved the process structure. We will define and use that calculus here.

3 Background

Our investigation of control flow is in the context of the π-calculus [7], a slightly modified syntax of which is given below:

$$P ::= \mathbf{0} \mid \bar{x}y.P \mid x(y \in Y).P \mid \tau.P \mid P + P \mid P|P \mid (\nu x)P$$

We borrow from [5] the idea of using a "filtered input" prefix rather than a match. The semantics of the filtered input prefix $x(y \in Y)$ dictate that – just as in the ordinary calculus – it communicate

$$Tau : \tau.P \xrightarrow{\tau} P$$

$$Out : \ \bar{a}b.P \xrightarrow{\bar{a}b} P \qquad In : \ a(y \in Y).P \xrightarrow{ab} P\{b/y\},\ b \in Y$$

$$Sum_0 : \ \frac{P_0 \xrightarrow{\mu} Q_0}{P_0 + P_1 \xrightarrow{\mu} Q_0 + \mathbf{0}} \qquad Sum_1 : \ \frac{P_1 \xrightarrow{\mu} Q_1}{P_0 + P_1 \xrightarrow{\mu} \mathbf{0} + Q_1}$$

$$Par_0 : \ \frac{P_0 \xrightarrow{\mu} Q_0}{P_0 | P_1 \xrightarrow{\mu} Q_0 | P_1} \qquad Res : \ \frac{P \xrightarrow{\mu} Q}{(\nu a)P \xrightarrow{\mu} (\nu a)Q},\ a \notin names(\mu)$$

$$Open : \ \frac{P \xrightarrow{\bar{a}b} Q}{(\nu b)P \xrightarrow{\bar{a}(b)} Q},\ b \neq a \qquad Close_0 : \ \frac{P_0 \xrightarrow{\bar{a}(b)} Q_0,\ P_1 \xrightarrow{ab} Q_1}{P_0 | P_1 \xrightarrow{\tau} (\nu b)(Q_0 | Q_1)}$$

$$Com_0 : \ \frac{P_0 \xrightarrow{\bar{a}(b)} Q_0,\ P_1 \xrightarrow{ab} Q_1}{P_0 | P_1 \xrightarrow{\tau} Q_0 | Q_1} \qquad Var : \ \frac{P' \equiv P \xrightarrow{\mu} Q \equiv Q'}{P' \xrightarrow{\mu} Q'}$$

Table 1: Operational semantics of the π-calculus taken from [5] (symmetric rules are ommitted).

with an output prefix of the form $\bar{x}z$ (for some z); however, in the filtered case, the communication occurs only when z is in the set Y. This yields an equally expressive calculus as the ordinary π-calculus with match, because the set Y could simply contain all names, and the match prefix $[x = y]P$ is encoded as $(\nu a)(\bar{a}x \mid a(y \in \{x\}).P)$.

The semantics used in [5] (reproduced in figure 1) are also modified from the standard calculus in that they do not eliminate nil processes, and do not allow associativity and commutativity of the composition and choice operators in their structural congruence in order to conserve the process' structure. Note that for the sake of simplicity, and for comparison with [5], we don't include replication in our language at this time. An unsophisticated treatment of replication could easily be added to our results, and we shall briefly discuss this in section 7.

The notion of "process addresses" was formalized through a localization operator $@\vartheta$ defined by induction on processes as follows (ϵ is the empty address):

1. $P@\epsilon = P$
2. $(P_0|P_1)@\|_i\vartheta = P_i@\vartheta$
3. $(P_0|P_1)@ +\!\!+_i\vartheta = P_i@\vartheta$
4. $\pi.P@\vartheta = ((\nu a)P)@\vartheta = P@\vartheta, \text{for } \vartheta \neq \epsilon$

This definition assigns to every program point an address. We define $Addr(P) = \{\vartheta \mid \exists Q : P@\vartheta = Q\}$ as the set of all addresses of a process P. Intuitively, two addresses are *compatible* (i.e. communication is possible between them) if they lay on the same side of a choice operator $+$, and on different sides of the same parallel composition operator $|$. This notion is defined in [5] as follows:

Definition 3.1. Given a process P and two addresses $\vartheta, \vartheta' \in Addr(P)$, ϑ and ϑ' are *compatible*, written as $\texttt{comp}_P(\vartheta, \vartheta')$, or symmetrically as $\texttt{comp}_P(\vartheta', \vartheta)$, if and only if

$$\vartheta = \vartheta_0 \|_i \vartheta_1 \text{ and } \vartheta' = \vartheta_0 \|_{1-i} \vartheta'_1, \ i \in \{0, 1\}$$

for some $\vartheta_1, \vartheta'_1$.

This addressing scheme encodes the ability of various program points in the π-calculus process to communicate. Therefore this representation will allow false positives computed by the flow insensitive analyses discussed above to be eliminated. The process representation that we build up will allow us to further improve the results obtained through this approach.

The work done by Nielson and Seidl [9] demonstrated that certain CFA's can be computed efficiently using the logical notion of Horn clauses with sharing, and we will exploit some of their results in order to maximize the running time of our final analysis. A discussion of the applicability and compositionality of our analysis will follow our main results.

4 Definitions

Given a π-calculus process P, over a countably infinite set of names $\mathcal{N}$, we list the basic definitions required for our representation and analysis below. Note that for the remainder of this paper, given any countable set S, we define $S_\perp$ as the flat domain over the elements of S with bottom element $\perp_S$, and the supremum of sets as their union unless otherwise indicated.

Definition 4.1. (Prefixes) An *output prefix* π_o is a pair (c, d) (for $c, d \in \mathcal{N}$). We say that c is the *channel* of π_o, and d is the *datum* of π_o. Similarly, we define an *input prefix* π_i as a triple (c, d, F) with $c, d \in \mathcal{N}$ and $F \in \wp(\mathcal{N})$. The set F is called the *filter* of π_i.

We define $\mathcal{O}_P$ and $\mathcal{I}_P$ as the set of output and input prefixes of a process P respectively, and $\Pi_P = \mathcal{O}_P \cup \mathcal{I}_P$ as the set of non silent prefixes of P. Furthermore, we'll use a few auxiliary definitions to handle our computations. First, we define our notion of the least upper bound of two functions:

Definition 4.2. (Supremum of Functions) Given functions $f : X \to Y$ and $g : X \to Y$ (with X,Y arbitrary sets), we define $f \sqcup g$ pointwise:

$$\forall x \in X.(f \sqcup g)(x) = f(x) \sqcup g(x)$$

We will also use the following definition of substitution on functions:

Definition 4.3. (Substitution on Functions) Given a function $f : X \to Y$, and given $x \in X$, and $y \in Y$, define $f[x \mapsto y]$ to be the same function as f, except that $(f[x \mapsto y])(x) = y$.

Next, we define binary tree forests as a set of nodes, and a pair of functions on each node yielding its left and right child. We first assume a countably infinite set of node names $\mathcal{V}$.

Definition 4.4. A *binary tree forest* F is a set of nodes $V_F \in \wp(\mathcal{V})$ paired with a pair of functions $r, l : \mathcal{V}_\perp \to \mathcal{V}_\perp$, i.e. $F = (V_F, (l, r))$, where $l(v)$ and $r(v)$ (for $v \in V_F$) are called the *left child* and *right child* of v respectively.

Now, letting $F = (V_F, (l, r))$ be a binary tree forest, we define the following elements of F:

Definition 4.5. A node $v \in V_F$ such that $l(v) = \bot_{\mathcal{V}}$ and $r(v) = \bot_{\mathcal{V}}$ is called a *leaf* of F. Informally, a leaf is a node with no children.

Definition 4.6. A node $v \in V_F$ such that $\nexists u \in V_F.(l(u) = v \vee r(u) = v)$ is called a *root* of F. Informally, a root is a node with no parents.

Definition 4.7. The *cardinality* of F is defined as the number of roots in F and is denoted $|F| = |\{v \in V_F | v \text{ is a root of } F\}|$. A forest of cardinality 1 is called a *tree*.

Now, we let $\bot_l = \bot_r = \bot_{\mathcal{V}_\bot \Rightarrow \mathcal{V}_\bot}$ denote the everywhere bottom function (i.e., the function that returns $\bot_{\mathcal{V}}$ on any input) in the domain of functions $\mathcal{V}_\bot \Rightarrow \mathcal{V}_\bot$. These elements are used as bottom elements for the left and right child functions (they indicate that no nodes have children).

Forests can be ordered into a complete lattice using the traditional subset ordering pointwise on their node sets and child functions:

Definition 4.8. Let $F_1 = (V_1, (l_1, r_1))$ and $F_2 = (V_2, (l_2, r_2))$ be forests,

$$\begin{aligned} F_1 \sqsubseteq F_2 &\Leftrightarrow (V_1 \subseteq V_2) \wedge (\forall v \in V_1 : (l_1(v) = l_2(v)) \wedge (r_1(v) = r_2(v))) \\ F_1 \sqcup F_2 &= (V_1 \cup V_2, (l_1 \sqcup l_2, r_1 \sqcup r_2)) \end{aligned}$$

This definition simply says that a forest is greater than any of its sub-forests, and that the least upper bound of two forests is defined as the union of their nodes paired with the least upper bound of their child functions. Let $\mathbb{F}$ denote the lattice of forests defined above, and let $\bot_{\mathbb{F}} = (\emptyset, (\bot_l, \bot_r))$. Note that, under specific conditions, the supremum of two single-tree forests will yield another single tree forest, and we will exploit this to generate a single tree representing a π-calculus process:

Proposition 4.1. *(Tree Preservation) Given forests $F_1 = (V_1, (l_1, r_1))$ and $F_2(V_2, (l_2, r_2))$ such $|F_1| = |F_2| = 1$ (i.e. F_1 and F_2 are trees). Then if*

1. *the root r of F_1 is a leaf of F_2 (or vice versa), and*
2. *$(V_1 \cap V_2) = \{r\}$*

then $(F_1 \sqcup F_2)$ is also a tree.

Proof. If the root of one tree is a leaf of the other, then taking the supremum operation merely grafts the latter tree onto the leaf of the former. We don't need to worry about the descendents or ancestors of the root/leaf node overlapping because we have assumed that the node sets are disjoint outside of this node. □

We now present various analyses of the π-calculus by first generating the representation of a process, and then using that representation to compute some conservative information about the behaviour of the process.

5 Process Representation

We begin by developing a representation that is equivalent to the one in [5] in that it allows us to determine what subprocesses can communicate with each other. Although the representations are equivalent, we describe our procedure in detail in order to make it evident how we achieve compositionality (discussed in section 7), and also how we can achieve more precise flow information.

The first step is to generate a tree that mimics the subprocess structure of the π-calculus process where each node represents the various subprocesses. We then generate a graph based on this representation that will denote which subprocesses can potentially communicate. Finally, we use this graph to compute our conservative estimate.

5.1 The Subprocess Tree

Given a π-calculus process P, we generate a triple (T, σ, ω) where T is a binary tree with subprocesses of P as nodes, and edges indicating a containment relation between subprocesses. The second component is a function $\sigma : V_T \to \wp(\Pi)$ indicating the actions (prefixes) that can occur at a particular subprocess address, and the third component $\omega : V_T \to \{\|, +\!\!+\}$ indicating that the topmost process operator at the given point (with $+\!\!+$ meaning choice, and $\|$ meaning composition). We adopt the convention that these functions take any node names not in their domains to bottom in their respective target domains in order to ensure that the join operations are well defined. We also assume that bound names are α-converted to avoid repetition, which we can easily do in a replication-free calculus.

We define the least upper bound of two such triples in the standard pointwise fashion. We define bottom elements $\bot_\sigma$ as the everywhere bottom function in the domain $\mathcal{V}_\bot \Rightarrow \wp(\Pi)_\bot$ (i.e. the domain of functions from nodes to sets of prefixes), and similarly $\bot_\omega$ as the everywhere bottom function in the domain $\mathcal{V}_\bot \Rightarrow \{\|, +\!\!+\}_\bot$.

We generate the triple (T, σ, ω) using the function $\mathcal{T}$ shown in table 2 defined on the π-calculus syntax. The name $v \in \mathcal{V}$ is initialized to a fresh node name that will denote the root of the tree, and a "capped" node name (like $\hat{v}$) denotes that the name should be fresh. We now show that this procedure has in fact constructed a single tree representing the π-calculus process:

Proposition 5.1. *Given a π-calculus process P, Let $\mathcal{T}[\![P]\!]_v = (T, \sigma, \omega)$. Then T is a tree rooted at v.*

Proof. The proof is by induction on the structure of P:

(Case $P \equiv \mathbf{0}$) The first component of the triple in this case is $(\{v\}, (\bot_l, \bot_r))$. This is a forest with only one node, and since the child functions are both bottom, the node has neither children nor parents. Thus it is certainly a tree rooted at v.

(Case $P \equiv x(y \in Y).Q$) By the induction hypothesis, we have that the first component of $\mathcal{T}[\![Q]\!]_v$ is a tree rooted at v. Call this tree $\hat{T}$. The first component of the triple $\mathcal{T}[\![P]\!]_v$ is $\hat{T} \sqcup \bot_{\mathbb{F}}$ by the definition of $\mathcal{T}$. Since we are taking the supremum of $\hat{T}$ and an empty tree, the result is simply $\hat{T}$ which is a tree rooted at v.

(Case $P \equiv \bar{x}y.Q$) This case is exactly analogous to the case for the input prefix.

$$
\begin{array}{rcl}
\mathcal{T}[\![\mathbf{0}]\!]_v & = & ((\{v\}, (\bot_l, \bot_r)), \bot_\sigma, \bot_\omega) \\
\mathcal{T}[\![x(y \in Y).P]\!]_v & = & \mathcal{T}[\![P]\!]_v \sqcup (\bot_{\mathbb{F}}, \bot_\sigma\ [v \mapsto (x, y, Y)], \bot_\omega) \\
\mathcal{T}[\![\overline{x}y.P]\!]_v & = & \mathcal{T}[\![P]\!]_v \sqcup (\bot_{\mathbf{F}}, \bot_\sigma\ [v \mapsto (x, y)], \bot_\omega) \\
\mathcal{T}[\![\tau.P]\!]_v & = & \mathcal{T}[\![P]\!]_v \\
\mathcal{T}[\![(\nu x)P]\!]_v & = & \mathcal{T}[\![P]\!]_v \\
\mathcal{T}[\![P_1 + P_2]\!]_v & = & \mathcal{T}[\![P_1]\!]_{\hat{v_1}} \sqcup \mathcal{T}[\![P_2]\!]_{\hat{v_2}} \sqcup \\
 & & ((\{v\}, (\bot_l\ [v \mapsto \hat{v_1}], \bot_r\ [v \mapsto \hat{v_2}])), \\
 & & \bot_\sigma, \bot_\omega\ [v \mapsto +\!\!+]) \\
\mathcal{T}[\![P_1 \mid P_2]\!]_v & = & \mathcal{T}[\![P_1]\!]_{\hat{v_1}} \sqcup \mathcal{T}[\![P_2]\!]_{\hat{v_2}} \sqcup \\
 & & ((\{v\}, (\bot_l\ [v \mapsto \hat{v_1}], \bot_r\ [v \mapsto \hat{v_2}])), \\
 & & \bot_\sigma, \bot_\omega\ [v \mapsto \|])
\end{array}
$$

Table 2: Constructing a tree to represent process P

(Case $P \equiv \tau.Q$) The result is immediate from the induction hypothesis on $\mathcal{T}[\![Q]\!]_v$.

(Case $P \equiv (\nu x)Q$) The result is immediate from the induction hypothesis on $\mathcal{T}[\![Q]\!]_v$.

(Case $P \equiv P_1 + P_2$) We have, by the induction hypothesis, that the first components of $\mathcal{T}[\![P_1]\!]_{\hat{v_1}}$ and $\mathcal{T}[\![P_2]\!]_{\hat{v_2}}$ are trees rooted at $\hat{v_1}$ and $\hat{v_2}$ respectively. Call these trees $\hat{T_1}$ and $\hat{T_2}$. It is trivial to verify that $T_v = (\{v\}, (\bot_l\ [v \mapsto \hat{v_1}], \bot_r\ [v \mapsto \hat{v_2}])$ is a tree rooted at v with $\hat{v_1}$ and $\hat{v_2}$ as leaves. By definition of $\mathcal{T}$, the first component of $\mathcal{T}[\![P_1 + P_2]\!]_v$ is $(T_v \sqcup \hat{T_1} \sqcup \hat{T_2})$, and since $\hat{v_1}$ and $\hat{v_2}$ are roots of $\hat{T_1}$ and $\hat{T_2}$, and leaves of T_v, proposition 4.1 yields the result that we are generating a tree rooted at v.

(Case $P \equiv P_1 \mid P_2$) This case works exactly analogously to the case for non deterministic choice.

□

The other two components of the generated triple are described as follows:

- The ω function encodes the information about whether each node's children are on opposite sides of a choice or a composition
- the σ function stores the set of top-level prefixes at a given node.

Note that a pair of composed (or "choiced") processes simply generate two new nodes, and two new edges, as well as adding information to the ω component about which operator is at the top level. Alternatively, leading prefixes cause no new information to be added to the tree or the ω

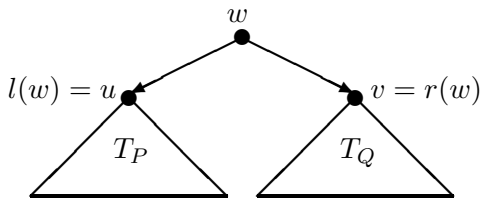


Figure 1: Compositional construction of T_{PQ} from T_P and T_Q

component, however, the prefix in question is added to the range of the current node in the σ function. In this way, the structure of the π-calculus process is recorded.

Observe that this construction is perfectly compositional in the sense that given trees $\mathcal{T}[\![P]\!]_u$ and $\mathcal{T}[\![Q]\!]_v$ for processes P and Q, we can build a tree for the combined processes $P \mid Q$ and $P + Q$ without recomputing the trees for P and Q as follows (see figure 1):

1. Generate a new node w to represent the combined process (and hence, the root of the combined tree)
2. Set the left child of w to the root of T_P, and the right child of w to the root of T_Q
3. Generate the combined functions σ_{PQ} and ω_{PQ} by computing the suprema of the respective functions, and then setting $\omega(w)$ appropriately.

This result is summarized as a corollary to the above proposition:

Corollary 5.2. *Given $\mathcal{T}[\![P]\!]_u = (T_P, \sigma_P, \omega_P)$ and $\mathcal{T}[\![Q]\!]_v = (T_Q, \sigma_Q, \omega_Q)$ for π-calculus processes P and Q, then $\mathcal{T}[\![P \mid Q]\!]_w = (T_{PQ}, \sigma_{PQ}, \omega_{PQ})$ can be computed as follows:*

$$\begin{aligned} T_{PQ} &= (\{w\}, (\bot_l\ [w \mapsto u], \bot_r\ [w \mapsto v])) \bigsqcup (T_P \sqcup T_Q) \\ \sigma_{PQ} &= (\sigma_P \sqcup \sigma_Q) \\ \omega_{PQ} &= (\omega_P \sqcup \omega_Q)[w \mapsto \|] \end{aligned}$$

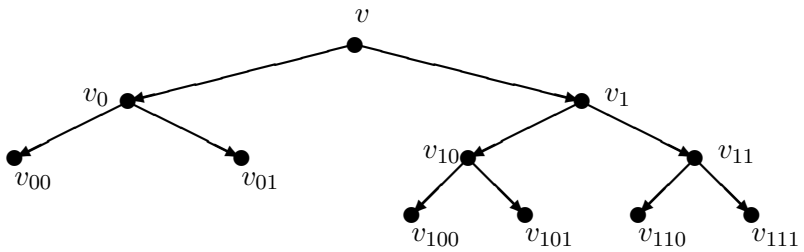
Proof. Immediate from proposition 5.1 and the definition of $\mathcal{T}$. □

Note that all of the operations above can be accomplished in constant time as the node names are completely distinct between the trees. An analogous result applies for the non-deterministic choice operator, but as we discuss in section 7, the parallel composition case is much more useful.

5.2 An Example of the Construction

We now give an example of the generation of such a representation for a given process. Consider the following π-calculus process:

$$a(x_1 \in \mathcal{N}).(\bar{b}x_1 + \bar{b}z) \mid (\bar{a}w.b(x_2 \in \mathcal{N}).(\bar{y}w.(\bar{a}w \mid \bar{a}y) \mid \bar{a}z.b(x_3 \in \mathcal{N}).(\bar{c}x_3 + \bar{c}z)))$$



Node	ω	σ	γ
v	$\parallel$	$\emptyset$	ε
v_0	$+\!\!+$	$\{(a, x_1, \mathcal{N})\}$	$\parallel_0$
v_1	$\parallel$	$\{(a, w), (b, x_2, \mathcal{N})\}$	$\parallel_1$
v_{00}	$\perp_\omega$	$\{(b, x_1)\}$	$\parallel_0 +\!\!+_0$
v_{01}	$\perp_\omega$	$\{(b, z)\}$	$\parallel_0 +\!\!+_1$
v_{10}	$\parallel$	$\{(y, w)\}$	$\parallel_1\parallel_0$
v_{11}	$+\!\!+$	$\{(a, z), (b, x_3, \mathcal{N})\}$	$\parallel_1\parallel_1$
v_{100}	$\perp_\omega$	$\{(a, w)\}$	$\parallel_1\parallel_0\parallel_0$
v_{101}	$\perp_\omega$	$\{(a, y)\}$	$\parallel_1\parallel_0\parallel_1$
v_{110}	$\perp_\omega$	$\{(c, x_3)\}$	$\parallel_1\parallel_1 +\!\!+_0$
v_{111}	$\perp_\omega$	$\{(c, z)\}$	$\parallel_1\parallel_1 +\!\!+_1$

Table 3: Tree generated by $\mathcal{T}$ on the example process

The tree generated on this example, as well as the values of the ω and σ functions is shown in table 3. The values of the l and r functions are indicated by the edges of the tree. Observe that we have carefully avoided repetition in our bound names as per the assumptions above. This is not necessary for the construction of the tree *per se*, but will be required for our analysis. Note that, as seen in the handling of nodes v_1 and v_{11} (representing the subprocesses $\bar{a}w.b(x_2 \in \mathcal{N}).(\ldots \mid \bar{a}y)$ and $\bar{a}z.b(x_4 \in \mathcal{N}).(\ldots)$ respectively), that sequences of prefixes result in the same tree structures as do single prefixes (the sequences are stored by the σ function). The γ entry simply shows the corresponding process address in the representation described in [5]. Each such address corresponds to a path in our tree.

5.2.1 Complexity of Tree Generation

Let N be the number of symbols in the process P. For every prefix, $\mathcal{T}$ only adds a single element to the set of prefixes associated with a given node (i.e., it adds an element to the image of one node v under the σ function), however since it is possible that a prefix is inserted into a given set multiple times (because of multiple occurrences of the same ouput prefix), the insertions may take linear time. The cases of the composition and choice operators add a single node and two edges to the tree, and add an element to the image of one node under the ω function. Since no node is a child of more than one other node (because we are generating a tree), and since each time edges are added to fresh nodes, we never add a node or an edge to a set more than once, and hence the insertions can be performed in constant time. Since there can be no more than a total of N prefixes, compositions and choice operations in a process P, the generation of the whole triple takes

at most quadratic time in the number of symbols in P.

5.3 The Communication Graph

We now use the tree created above to generate a graph indicating which subprocesses can communicate with one another. This will be achieved by augmenting the tree structure describing the process with edges indicating potential communications. We take the conservative initial assumption that each prefix could potentially communicate, thereby allowing the subprocesses rooted at the prefix's node to potentially communicate as well. Note that although this approach ensures a conservative result, it also limits precision. A re-examination of this assumption will permit greater accuracy in results, and is described in section 6. Initially, however, assume that all prefixes could potentially communicate.

In order to determine the communication ability of subprocesses, we need only observe the following:

1. prefixes at a particular node can never communicate with prefixes at any of that node's descendants (because we have not included replication in our syntax)
2. the left child (and all of its descendants) of a composition node (as given by the ω function) may communicate with the right child and all of its descendants (and vice versa)
3. neither the children of choice nodes (i.e. nodes v such that $\omega(v) = +\!\!+$) nor any of their descendants may communicate with each other

5.3.1 Graph Construction

We now proceed with the construction of the graph $\mathcal{G}$ by iterating on the structure of the tree T. To do so, we first define an auxiliary function $\delta_\tau : \mathcal{V}_\perp \to \wp(\mathcal{V})_\perp$ to compute the set of descendants of a node v in τ, where τ is a triple (T, σ, ω) as defined above. We define δ_τ recursively as follows:

$$\delta_\tau(v) = \begin{cases} \{v\} \bigcup (\delta_\tau(l(v)) \cup \delta_\tau(r(v))) & \text{if } v \neq \perp_{\mathcal{V}} \\ \emptyset & \text{otherwise} \end{cases}$$

Note that the time complexity of δ_τ is linear in the number of symbols in the original process P. This is simply because at each step of the recursion, only a single node is added to the list, and since no node is ever added twice (because there are no back edges in the tree), each union operation takes only constant time. Since there can only be a maximum of N subprocesses in P, the whole operation is linear in N.

We can now use this function to define our graph $\mathcal{G}$. First we take the convention that an "edge" (U, V) where V and U are sets of nodes is defined as the set of edges between all the nodes in U and V, or more formally (and simply) as $U \times V$.

Again, letting $\tau = \mathcal{T}[\![P]\!]_v = (T, \sigma, \omega)$, where the tree T has root node v. We now define the function $\mathcal{E}_\tau$ as the following function (taking a node v as input, and generating a set of edges):

$$\mathcal{E}_\tau(v) = \begin{cases} (\delta_\tau(l(v)), \delta_\tau(r(v))) \bigcup \\ (\mathcal{E}_\tau(l(v)) \cup \mathcal{E}_\tau(r(v))) & \text{if } (v \neq \perp_{\mathcal{V}}) \wedge (\omega(v) = \|) \\ \\ (\mathcal{E}_\tau(l(v)) \cup \mathcal{E}_\tau(r(v))) & \text{if } (v \neq \perp_{\mathcal{V}}) \wedge (\omega(v) = +\!\!+) \\ \\ \emptyset & \text{otherwise} \end{cases}$$

The complete communication graph is computed as $\mathcal{G} = (V_T, \mathcal{E}_\tau(v))$. Observe that if the given node v is a composition node (i.e. $\omega(v) = \|$), then edges are added to form a complete bipartite graph between the left and right subtrees of that node (i.e. $\delta_\tau(l(v))$ and $\delta_\tau(r(v))$) and these edges are added recursively to the edges resulting from calling the function on each of those subtrees. If the given node is a choice, then no edges are added between the two subtrees at that level, and only the union of the edges resulting from calling the functions recursively on the subtrees is returned.

As in the case for the tree generation, the structure of this algorithm allows the graph representation to also be constructed compositionally:

Proposition 5.3. *Given π-calculus processes P and Q with corresponding triples $\tau_1 = \mathcal{T}[\![P]\!]_u = (T_P, \sigma_P, \omega_P)$ and $\tau_2 = \mathcal{T}[\![Q]\!]_v = (T_Q, \sigma_Q, \omega_Q)$, and graph representations $G_P = (V_P, \mathcal{E}_{\tau_1}(u))$ and $G_Q = (V_Q, \mathcal{E}_{\tau_2}(v))$, then we can construct the combined graph $G_{PQ} = (V_{PQ}, E_{PQ})$ for the process $P \mid Q$ as follows:*

$$\begin{aligned} V_{PQ} &= V(T_{PQ}) \\ E_{PQ} &= (\delta_{\tau_1}(u), \delta_{\tau_2}(v)) \bigcup (E_P \cup E_Q) \end{aligned}$$

with $V(T_{PQ})$ denoting the vertex set of the tree T_{PQ} computed as in corollary 5.2.

Proof. Follows immediately from the definition of $\mathcal{E}_\tau$. □

Thus, the only new information in the overall graph that we need to compute are the edges *between* the nodes in the given graphs. As in the case for the tree, the analogous result also applies in the case of the choice operator.

5.3.2 Equivalence of Graph Model

Now, we compare this to the representation defined in [5], which we reproduced in section 3. We first define a mapping from the nodes in our tree (which represent process points) and addresses in Bodei et al's representation. For a given process P it is trivial to see that each address $\vartheta \in Addr(P)$ corresponds in a one-to-one way to a path in the tree $\tau = \mathcal{T}[\![P]\!]_v$, thus we can generate the requisite mapping $\gamma : V_T \to Addr(P)$ by building it through a simple tree traversal:

$$\gamma_\tau(v) = \begin{cases} \epsilon & \text{if } v \text{ is the root node} \\ \\ \gamma_\tau(p)\|_0 & \text{if } (l(p) = v) \wedge (\omega(p) = \|) \\ \gamma_\tau(p)\|_1 & \text{if } (r(p) = v) \wedge (\omega(p) = \|) \\ \\ \gamma_\tau(p) +\!\!+_0 & \text{if } (l(p) = v) \wedge (\omega(p) = +\!\!+) \\ \gamma_\tau(p) +\!\!+_1 & \text{if } (r(p) = v) \wedge (\omega(p) = +\!\!+) \end{cases}$$

In order to show that the two representations are truly equivalent, we must demonstrate that they allow the same subprocesses to communicate. The following lemma demonstrates just this property:

Lemma 5.4. *For any given π-calculus process P, let $\mathcal{G}[\![P]\!] = (V_\mathcal{G}, E_\mathcal{G})$ be the graph generated by the procedure outlined above, and let $\gamma : V_\mathcal{G} \to Addr(P)$ assign each node to its corresponding address. Then for any pair of nodes $u, v \in V_\mathcal{G}$, we have*

$$(u, v) \in E_\mathcal{G} \text{ if and only if } \texttt{comp}_P(\gamma(u), \gamma(v))$$

Proof. Suppose that $(u, v) \in E_\mathcal{G}$, then $\exists n \in V_\mathcal{G}$ such that $\omega(n) = \|$ and, without loss of generality, $u \in \delta_\tau(l(n)) \ \wedge \ v \in \delta_\tau(r(n))$ (for τ the triple generated for P by $\mathcal{T}$) by the definition of $\mathcal{E}_\tau$. Since n is a composition node ($\omega(n) = \|$), then $\gamma(l(n)) = \vartheta_0\|_0$ and $\gamma(r(n)) = \vartheta_0\|_1$ for some ϑ_0 by the definition of $@\vartheta$ and γ, and subsequently any descendants of these children will share this prefix in their address. Thus $\gamma(u) = \vartheta_0\|_0\vartheta$ and $\gamma(v) = \vartheta_0\|_1\vartheta'$ for some ϑ, ϑ'. Thus, by the definition of $\texttt{comp}_P$, we have $\texttt{comp}_P(\gamma(u), \gamma(v))$.

Conversely, suppose that $\texttt{comp}_P(\gamma(u), \gamma(v))$, we then have that $\gamma(u) = \vartheta_0\|_0\vartheta$ and $\gamma(v) = \vartheta_0\|_1\vartheta'$ for some $\vartheta, \vartheta', \vartheta_0$ by the definition of $\texttt{comp}_P$. Since $\gamma(u)$ and $\gamma(v)$ share a prefix (ϑ_0), v and u must ergo share an ancestor n such that $\gamma(n) = \vartheta_0$. We know from $\gamma(u)$ and $\gamma(v)$ that n must be a composition node (i.e. $\omega(n) = \|$) because the next components of the addresses are $\|_0$ and $\|_1$. Thus $u \in \delta_\tau(l(n))$, $v \in \delta_\tau(r(n))$ (for τ as above), and $\omega(n) = \|$, thus $(u, v) \in E_\mathcal{G}$ by line the definition of $\mathcal{E}$.

□

5.3.3 Complexity of Graph Generation

We again let N denote the number of symbols in our π-calculus process P. Letting $\tau = \mathcal{T}[\![P]\!]_v = (T, \sigma, \omega)$ be the triple generated by the procedure outlined in section 5.1, we proceed to analyze the complexity of the edge generation function $\mathcal{E}_\tau$. Note that in the worst case, every node encountered will be a composition node requiring edges to be added. At each recursive step, each call to δ_τ requires linear time in N as noted above, and at most N^2 edges are added between the two subtrees (because each subtree can contain at most N nodes). Since no edge is ever added twice (because each recursive call adds edges within a subtree within which no edges previously existed due to the bipartite condition above), the union operations take only constant time. Finally, since there can be at most N recursive calls (because there can be at most N composition operators), the computation of the edge set can take at most $O(N^3)$ time.

6 An Incremental Approach

Thanks to lemma 5.4, we could trivially modify the flow logic presented in [5] to use the presence of an edge in our graph rather than the addressing scheme in order to achieve the same analysis. However, the primary advantage of this representation is that it can be built incrementally in order to obtain more accurate results. The key to doing so lies in the observation that not all of the prefixes of a π-calculus process P can initially communicate, even if they lie on opposite sides of a composition operator. For example, consider the process $P \equiv a(x \in \mathcal{N}).(P_1|P_2) \mid \bar{b}y.(P_3|P_4)$. The process representation we have thus far would conservatively compute that the subprocesses P_1,

$(R,K) \models \mathbf{0}$	iff	*true*
$(R,K) \models \tau.P$	iff	$(R,K) \models P$
$(R,K) \models \bar{x}y.P$	iff	$(R,K) \models P \wedge$ $\forall u \in R(x) : R(y) \subseteq K(u)$
$(R,K) \models x(y \in Y).P$	iff	$(R,K) \models P \wedge$ $\forall u \in R(x) :$ $(K(u) \cap R(Y)) \neq \emptyset \Rightarrow (K(u) \cap R(Y)) \subseteq R(y)$
$(R,K) \models P_1 + P_2$	iff	$(R,K) \models P_1 \ \wedge \ (R,K) \models P_2$
$(R,K) \models P_1 \mid P_2$	iff	$(R,K) \models P_1 \ \wedge \ (R,K) \models P_2$
$(R,K) \models (\nu x)P$	iff	$(R,K) \models P$

Table 4: Flow Logic for the π-calculus (modified from [9])

P_2, P_3, and P_4 are in fact compatible with one another. However, this is clearly not the case, as it is impossible for these subprocesses to communicate until the leading prefixes $a(x \in \mathcal{N})$ and $\bar{b}y$ communicate. The reason for the innaccuracy is that our current representation implicitly assumes that it is possible for these prefixes to communicate. This can be rectified by paring down the edge set that we have generated in our graph model to reflect the presence of leading prefixes at a program point, and proceeding to do an initial analysis on this pruned model. The results of this analysis will allow us to determine whether it is possible for these prefixes to indeed communicate during execution, and if so to reintroduce the appropriate edges reflecting the new state in order to refine our analysis in the next step.

6.1 Horn Clauses and Flow Insensitive Analysis

Our approach uses the flow *insensitive* analysis discussed in section 3 as a building block for writing a more accurate analysis. In particular, we'll use techniques introduced by Nielson and Seidl in [9] using systems of HCS's (Horn Clauses with Sharing) in order to maximize the efficiency of our analysis. In the latter, it was shown that a flow insensitive analysis equivalent to those in [4] and [14] could be computed in time cubic in the size of the process. We present in table 4 a slight modification of the flow logic presented in [9] that computes an equivalent analysis for our version of the calculus (in which match and replication are omitted, but filtered inputs are included).

Let β_P represent the set of names of the process P that are *bound by input*, i.e. they appear as the datum in an input prefix in P, and let χ_P represent all of the other names in P. The result of the analysis is a pair (R,K), where $R : \beta_P \to \wp(\chi_P)$ records information about the set of names to which a given name can be bound (via an input prefix), and $K : \chi_P \to \wp(\chi_P)$ records information about the set of names that can be transmitted over a given name (via an output prefix). We

$$
\begin{array}{rcl}
\mathcal{H}[\![\mathbf{0}]\!] & = & \mathbf{1} \\
\mathcal{H}[\![\overline{x}y.P]\!] & = & \mathcal{H}[\![P]\!] \wedge \\
 & & \forall u : \forall v : (R(u,x) \wedge R(v,y)) \Rightarrow K(v,u) \\
\mathcal{H}[\![x(y \in Y).P]\!] & = & \mathcal{H}[\![P]\!] \wedge \\
 & & \forall u : \forall v : \forall w : \\
 & & (R(u,x) \wedge Y(v) \wedge K(w,u) \wedge R(w,v)) \Rightarrow R(w,y) \\
\mathcal{H}[\![\tau.P]\!] & = & \mathcal{H}[\![P]\!] \\
\mathcal{H}[\![(\nu x)P]\!] & = & \mathcal{H}[\![P]\!] \\
\mathcal{H}[\![P_1 + P_2]\!] & = & \mathcal{H}[\![P_1]\!] \wedge \mathcal{H}[\![P_2]\!] \\
\mathcal{H}[\![P_1 \mid P_2]\!] & = & \mathcal{H}[\![P_1]\!] \wedge \mathcal{H}[\![P_2]\!]
\end{array}
$$

Table 5: Horn Clauses with Sharing for the π-calculus (modified from [9])

generalize these functions to sets in the obvious way, and (as in [9, 4]) we adopt the convention that $\forall x \in \chi_P.R(x) = x$[1].

The flow logic imposes a set of constraints on these functions which depend on the prefixes encountered while decomposing P. For instance, upon encountering an output prefix $\bar{x}y$, it is asked that for any name $u \in R(x)$ (i.e. any name that x could assume), that $R(y)$ be contained in $K(u)$ (i.e. that the names to which y could be bound are contained in the set of names that could be transmitted over each u). This exactly captures the behaviour of the output prefix. The more complicated case of the filtered input $x(y \in Y)$ generates the following constraint:

$$\forall u \in R(x) : (K(u) \cap R(Y)) \neq \emptyset \Rightarrow (K(u) \cap R(Y)) \subseteq R(y)$$

This says that for any name u to which x could be bound, a non-empty intersection of $K(u)$ and $R(Y)$ (indicating the possibility that whatever is received on x could be a member of Y) implies that that very intersection should be contained in $R(y)$ (the set of possible bindings for the name y).

We would like to generate HCS's corresponding to our flow logic. Following [9], we express set inclusions of the form $X \subseteq Y$ using set memberships of the form $\forall u : u \in X \Rightarrow u \in Y$, set memberships of the form $u \in R(v)$ using a binary predicate of the form $R(u,v)$, and set memberships of the form $u \in Y$ using a binary predicate of the form $Y(u)$. Using these conventions, it can easily be checked that our flow logic formulation is logically equivalent to the HCS formulation shown in table 5 (with the caveat that we conjuct clauses of the form $R(x,x)$ for every $x \in \chi_P$ in order to enforce the added convention on the function R).

[1] Intuitively, this captures the notion that a name not bound by input cannot be anything other than itself

Using the techniques presented in [9], these clauses can be solved in polynomial time in order to arrive at the final solution (R, K). This efficiency is due to the result from [9] stating that an HCS system can be solved in polynomial time indexed by the maximum nesting depth of quantification in any clause, which in our case is 3. We now proceed to use this solver in order to build a more accurate solution using an incremental algorithm.

6.2 Accuracy in Steps

We observe that the Horn clause system $\mathcal{H}$ above only generates new clauses when an input or output prefix is encountered (excluding the auxiliary clauses generated for names not bound by input). We can thus define the solver for the system as a function $\mathcal{HCS}$ which takes a set of prefixes to a pair of functions of the form (R, K) by taking the following steps:

1: **Input**: a set of prefixes S
2: for each prefix $\pi \in S$ generate the clause $\mathcal{H}(\pi)$
3: for each name $x \in \pi$ not bound by input generate the clause $R(x, x)$
4: solve the conjunction of the clauses to obtain a solution (R, K)

Next, we modify our graph generation function from section 5.3.1 to only generate the edges required in a given state. In order to do so, we must know an intermediate solution of our analysis so that we can determine which prefixes may have communicated.

Given a π-calculus process P, let $\tau = \mathcal{T}[\![P]\!]_v = (T, \sigma, \omega)$ as defined in section 5.1. In order to keep track of whether prefixes have communicated, we update a context function $\mathcal{C} : (V_T \times \Pi_P) \rightarrow \{\texttt{tt}, \texttt{ff}\}$ (recalling that Π_P is the set of non-silent prefixes of process P). Given a node v and a prefix, $\mathcal{C}$ returns `tt` if it has been determined that the given prefix is in $\sigma(v)$ and could communicate during execution (according to the intermediate solution), and `ff` otherwise. In other words, we require the function $\mathcal{C}$ to preserve the following property:

Property 1. Given a triple $\tau = (T, \sigma, \omega)$ generated from a process P, and an intermediate flow-insensitive solution (R, K) generated by $\mathcal{HCS}$ on some set of prefixes $S \subseteq \Pi_P$, then for any prefix π in S and node $v \in V_T$, $\mathcal{C}(v, \pi) = \texttt{tt}$ if and only if:

1. $\pi \in \sigma(v)$, and
2. $\forall n \in \mathcal{N} : (K(n) \neq \emptyset) \Rightarrow (n \in R(\mathit{channel}(\pi)))$

The property says that prefix π in node v has communicated if some name n over which a value was transmitted (given by $K(n)$) is included in the set of things that its channel could be (given by the R function). Note that if π is an input prefix, then this would imply that it could potentially receive the value over n. If π is an output prefix, then $R(\mathit{channel}(\pi))$ is trivially $\mathit{channel}(\pi)$, and the property simply states that this could be a prefix that sent the value.

In order to restrict the edges that are added, we first introduce a binary predicate on the function $\mathcal{C}$. Given $\mathcal{C}$, and node $v \in V_T$ (as given by τ), we define the predicate $\psi_\tau(v, \mathcal{C})$ as follows:

$$\psi_\tau(v, \mathcal{C}) = ((\sigma(v) = \emptyset) \ \vee \ (\forall \pi \in \sigma(v) : \mathcal{C}(v, \pi) = \texttt{tt}))$$

This predicate is true either when the set of leading prefixes at node v (i.e. $\sigma(v)$) is empty, or when the $\mathcal{C}$ function tells us that it has been determined that they can all communicate. This captures our notion that we only want subprocesses without interfering prefixes to be able to communicate, and will thus be used extensively in the following.

Using ψ_τ, we can define a more restrictive version of the δ_τ function defined in section 5.3. There, we defined $\delta_\tau(v)$ to be the set of descendants of the node v in τ, we will now define a very similar function $\delta'_\tau(v, \mathcal{C})$ which will return only those descendents that are not behind leading prefixes that cannot communicate according to the function $\mathcal{C}$. The funciton is defined as follows:

$$\delta'_\tau(v,\mathcal{C}) = \begin{cases} \{v\} \bigcup (\delta'_\tau(l(v),\mathcal{C}) \cup \delta'_\tau(r(v),\mathcal{C})) & \text{if } v \neq \perp_{\mathcal{V}} \wedge \psi_\tau(v,\mathcal{C}) \\ \emptyset & \text{otherwise} \end{cases}$$

Observe that this algorithm is identical to the δ_τ function except for the additional condition ψ_τ in the recursive step. This requires either that the set of prefixes at the current node (i.e., $\sigma(v)$) be empty in which case nothing is inhibiting the sub-processes from communicating normally, or that it has been determined that every prefix in the set could possibly communicate (according to the supplied $\mathcal{C}$ function) at some point during execution. It is clear that this function returns a subset of the nodes returned by the original function δ_τ.

We similarily define a more restrictive edge generation function $\mathcal{E}'_\tau$ as follows:

$$\mathcal{E}'_\tau(v,\mathcal{C}) = \begin{cases} (\delta'_\tau(l(v),\mathcal{C}), \delta'_\tau(r(v),\mathcal{C})) \bigcup (\mathcal{E}'_\tau(l(v),\mathcal{C}) \cup \mathcal{E}'_\tau(r(v),\mathcal{C})) & \text{if } (v \neq \perp_{\mathcal{V}}) \wedge (\omega(v) = \|) \wedge \psi_\tau(v,\mathcal{C}) \\ (\mathcal{E}'_\tau(l(v),\mathcal{C}) \cup \mathcal{E}'_\tau(r(v),\mathcal{C})) & \text{if } (v \neq \perp_{\mathcal{V}}) \wedge (\omega(v) = +\!\!+) \wedge \psi_\tau(v,\mathcal{C}) \\ \emptyset & \text{otherwise} \end{cases}$$

Note that, just like the modification we made to the δ_τ function, $\mathcal{E}'_\tau$ is identical to $\mathcal{E}_\tau$ except for the additional restriction on the two recursive possibilities (and the fact that δ'_τ is used to compute descendents rather than δ_τ). In fact, we notice that if $\mathcal{C}$ were to return true on every pair of inputs, then the predicate ψ_τ would trivially evaluate to true and the function $\mathcal{E}'_\tau$ would be identical to the function $\mathcal{E}_\tau$. It is therefore similarly obvious that this function generates no more edges than the original function $\mathcal{E}_\tau$. We state this result without proof as a lemma:

Lemma 6.1. *For any v, τ, and $\mathcal{C}$ we have*

$$\mathcal{E}'_\tau(v,\mathcal{C}) \subseteq \mathcal{E}'_\tau(v, \lambda xy.\mathtt{tt}) = \mathcal{E}_\tau(v)$$

Observe that the polynomial complexity of the function has at worst become $O(N^4)$ as the added condition can be checked in linear time (implying that the δ'_τ function is now quadratic). Note also that a similar compositionality property to that proved in proposition 5.3 holds here because combining two given π-calculus processes using a composition or choice operator does not create leading prefixes in the combined process, thus the added condition of ψ_τ will trivially succeed at the top level of the overall process.

We can now use this algorithm to define our full incremental CFA for the π-calculus. Given a node v, define $n_G(v) = \{u \in V_G | (u, v) \in E_G\} \cup \{v\}$ as the *neighborhood* of v including v itself. We'll also extend the domain of the function σ to sets in the obvious way.

Algorithm 1 Incremental Analysis on π-calculus

```
1: CFA(P) ≡
2: Initialize: τ = (T, σ, ω) = T[[P]]_v
3: Initialize: C to return ff on all input
4: Initialize: R and K functions to return ∅ on all input
5: repeat
6:    Compute G = (V_T, E'_τ(v, C))
7:    for all nodes u ∈ E_G such that n_G(u) ≠ ∅ do
8:       Compute S = σ(n_G(u))
9:       Compute (R', K') = HCS(S)
10:      Set (R, K) ← (R ⊔ R', K ⊔ K')
11:      for all names n such that K'(n) ≠ ∅ do
12:         for all π ∈ σ(u) such that n ∈ R'(channel(π)) do
13:            C(u, π) ← tt
14:         end for
15:      end for
16:   end for
17: until G does not change
18: return (R, K)
```

Our full algorithm is presented as algorithm 1. The procedure is initialized by generating the triple $\tau = (T, \sigma, \omega)$ for the given process (line 4), and setting the $\mathcal{C}$ and initial solution to bottom. Then the following steps are iterated:

1. Line 6: A graph G is generated based on the communication assumptions given by the function $\mathcal{C}$ (initially, it is assumed that no prefixes in the process can communicate).
2. Lines 7-9: For every node in G, an intermediate solution (R', K') is computed by running the flow-insensitive solver $\mathcal{HCS}$ on the set of prefixes in the node's neighborhood
3. Line 10: The overall solution (R, K) is updated with the results in the intermediate solution
4. Lines 11-15: The communication function $\mathcal{C}$ is updated with the results of the intermediate solution. For any name that can potentially transmit a value (line 11), all leading prefixes at the current subprocess that could potentially have n as a channel during execution (line 12), are marked as potentially able to communicate (line 13).

These steps are iterated until the graph G ceases to change. It is clear that the iterated function is monotone, as no iteration ever sets any pair in the domain of $\mathcal{C}$ to `ff`, thus no edges are ever removed from G between iterations. Elementary fixed point theory tells us that this function will terminate. Since G can have at most $O(N^2)$ edges (with N the number of symbols in P), it can only be iterated at most this many times. Since the edge generation function $\mathcal{E}'_\tau$ (time $O(N^4)$), the $\mathcal{HCS}$ solver (time $O(N^4)$), and the computation of the suprema of R and K (achievable through N set merges for a total time of $O(N^3)$) are all polynomial time functions iterated $O(N^2)$ times in the inner loop, and another $O(N^2)$ in the outer loop, we can safely conclude that the overall

analysis also runs in $O(N^8)$ time, polynomial in the number of symbols in P. However, this is a very coarse performance analysis, and we discuss the possibility of improving it in section 7.

The correctness of the algorithm in this setting means that no potential communications in P are left out of the solution (R, K). This follows from from the correctness of $\mathcal{HCS}$ (whose solution includes all of the potential communications between the prefixes in its input set), if we can establish that the edges computed by the function $\mathcal{E}'_\tau$ include all the edges that could communicate. But by lemma 6.1, this is upper bounded by the edges computed by $\mathcal{E}_\tau$, any restrictions made by $\mathcal{E}'_\tau$ are contingent on the predicate ψ_τ whose correctness only depends on the fact that the communication function $\mathcal{C}$ adheres to property 1. Thus we can prove the correctness of our algorithm by showing that this property is invariant in algorithm 1:

Theorem 6.2. *Property 1 on the function $\mathcal{C}$ is invariant in $\mathcal{CFA}(P)$ for any process P.*

Proof. Initially (i.e., before the loop of lines 5-17), $\mathcal{C}(v, \pi) = \texttt{ff}$ for all v and π and $K(n) = \emptyset$ for all names, thus the property trivially holds. Inside the loop, $\mathcal{C}$ is only updated at line 13. Here the loop conditions of lines 11 imposes the condition that the name n has $K'(n) \neq \emptyset$ giving us the antecedent of the second condition of property 1. The loop condition on line 12 requires that the updated prefix π be contained in $\sigma(u)$ (where u is the node updated) satisfying condition 1 of property 1, and that n be contained in $R'(\mathit{channel}(\pi))$ yielding condition 2. Hence the update of $\mathcal{C}$ at line 13 does not change the invariant that property 1 is satisfied yielding the correctness of the algorithm. □

Furthermore, we make the observation that the set S of prefixes computed in line 8 is a subset of the set of prefixes of the entire process, i.e. $S \subseteq \Pi_P$ in every iteration. By the monotonicity of the flow-insensitive version of the analysis we have that $\mathcal{HCS}(S) \subseteq \mathcal{HCS}(\Pi_P)$ thereby immediately establishing that our solution is at least as accurate as the flow-insensitive version. Furthermore lemma 6.1, which establishes an upper bound on the edges in our graph under the conservative assumption that all prefixes can communicate, and lemma 5.4 yield that our analysis is at least as accurate as a similar result computed on the (non-incremental) addressing model of [5]. The results of an analysis on such a model can actually be computed by using the edge set computed by $\mathcal{E}_\tau$ rather than $\mathcal{E}'_\tau$ in our algorithm. Let $\texttt{CFA}_{\mathcal{E}'}$ denote the incremental algorithm and $\texttt{CFA}_{\mathcal{E}}$ denote the more conservative result, and let $P \equiv \bar{a}b.(\bar{x}y|x(z))$. We can easily check that $\texttt{CFA}_{\mathcal{E}}(P)$ computes that $K(x) = \{y\}$ and $R(z) = \{y\}$ and the empty set for all other names, whereas $\texttt{CFA}_{\mathcal{E}'}(P)$ computes that K and R are empty for all names (because the leading prefix $\bar{a}b$ has no opportunity to communicate). These observations together imply our main results:

Theorem 6.3. *Let $\sqsubseteq$ be the standard inclusion ordering taken pointwise on the lattice of all possible functions from processes P to solutions (R, K) of our analysis, then we have*

$$\texttt{CFA}_{\mathcal{E}'} \sqsubset \texttt{CFA}_{\mathcal{E}}$$

Proof. As noted, the result follows from lemmas 6.1 and 5.4 yielding that the incremental result is *at least* as accurate. The domain of the function is a lattice because we are using an orderable graph representation of the processes. The effect of the two algorithms on the above process P demonstrates that $\texttt{CFA}_{\mathcal{E}'}$ is actually strictly more accurate. □

7 Discussion and Future Work

The analysis we have presented improves the accuracy of previous results by considering leading prefixes of sub-processes in a more sophisticated way. This granularity of analysis can be beneficial when it is necessary to establish temporal properties of processes to verify a desired security policy. Specifically in the π-calculus, our analysis establishes that certain preliminary communications take place *before* other computations are permitted. Such information can be used to avoid falsely identifying handshaking operations like password checks and key exchanges as information leakage. This level of accuracy is the primary advantage of our analysis over previous approaches. However, there is another property of our analysis that yields a good possibility that it could eventually be used in practice: and that is its inherent compositionality. We discuss this property of our representation here, and also touch on the possibility of adding replication to our language.

7.1 Compositionality

Suppose that we have a π-calculus process S that acts as a server, and a family of client processes $\{C_i\}$. In a realistic situation, an information flow analysis on any of these individual processes is of limited use. It is much more useful if a control flow analysis is done on the composition of the server with any client process that may request to interact with it at a given time, i.e. the security policy should in general be checked on the combined process $S \mid C$ for any client C that wishes to communicate with S. A static analysis on this process can be used to determine whether it is safe (with respect to a particular security policy) for the server to interact with the client. Our approach allows us to pre-compute a process representation (i.e. the communication tree and graph) for each process and to compose them if an analysis is ever required. Given the pre-computed tree representations of the client and server processes, the compositionality properties of our representation (corollary 5.2 and proposition 5.3) allows us to efficiently compute the corresponding representation for their interaction. However, it is not clear if a similarly desireable property holds of the actual results of the analysis of the subprocesses, as each prefix could potentially be able to communicate with all of the prefixes of the other subprocess, perhaps requiring the recomputation of the data. A compositionality result would mean that our analysis would be *fully* compositional, drastically increasing its practical feasability. We conjecture that full compositionality can be achieved, but leave its proof for future work.

7.2 A Note on Replication

Thus far, we have not added the standard replication construct in our treatment of the π-calculus. However, a naïve treatment of the issue is actually quite trivial to include. A simple conservative approach to replication is to assume that a replicated subprocess can communicate with every subprocess, including itself and its descendants, but excluding those subprocesses that lie on the opposite side of a choice operator from itself. The same rule would also need to be applied to the descendents of the node corresponding to the subprocess. This latter fact can plainly be seen by example thanks to processes such as $P \mid !(\bar{x}y.(Q \mid R))$ where it can be seen that such a conservative estimate must account for the fact that Q and R may communicate with P, as well as with the prefix $\bar{x}y$ itself. We have left this conservative assumption out of our analysis at this time because it would have complicated the presentation without providing any new insight. A more sophisticated method for dealing with the replication operator involving techniques such as the careful consideration of

leading prefixes that we have introduced here is certainly plausible, but left for future work.

7.3 Efficiency

Finally, we note that the running time of our analysis as given, while polynomial may be quite high: the most elementary analysis of the running time of the algorithm yields an $O(N^8)$ worst-case performance. It is expected that an improvement in accuracy over previous results may lead to a decrease in efficiency, but we also conjecture that careful analysis of the our algorithm can significantly reduce this number in the average case. The rationale is that our worst-case assumptions are based on the presence of $O(N)$ subprocesses *and* $O(N)$ prefixes per subprocess at the same time, quantities that are inversely proportional in the average case. Furthermore, we observe that our algorithm computes a lot of redundant information: for instance, every intermediate result from the $\mathcal{HCS}$ algorithm comes from considering the set of prefixes in the neighborhood of a node, and this is repreated for each node. This means that the interaction between the nodes at the endpoints of each edge are actually re-computed (once for each endpoint). It may be possible to pre-compute edge information in order to further reduce the worst-case complexity of the analysis.

8 Conclusions

We have developed a new control flow analysis for the π-calculus that improves the accuracy of previous results by considering the behavior of leading prefixes over possible executions of processes. We have also shown that our approach is at least partially compositional, in the sense that our representational structure can efficiently be reused when analysis of processes is needed in different contexts.

While our analysis as a whole has a high polynomial running time, we have indicated ways in which the analysis could be refined in order to reduce this. Furthermore, we have conjectured a compositionality result on the results of our analysis that would improve its feasibility in practice if it were proven to be true.

We have not considered the behaviour of the replication construct in our analysis, but have discussed how it could be handled conservatively. A more sophisticated approach to replication, as well as further improvements to the accuracy of the analysis have been left for future work.

References

[1] 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.

[2] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

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

[4] C.Bodei, P.Degano, F.Nielson, and H.Riis Nielson. Control flow analysis for the π-calculus. In *Proceedings of CONCUR '98*, volume 1466 of *Lecture Notes In Computer Science*, pages 84–98. Springer-Verlag, 1998.

[5] C.Bodei, P.Degano, C. Priami, and N. Zannone. An enhanced CFA for security policies. In *Proceedings of the Workshop on Issues on the Theory of Security (WITS '03), (co-located with ETAPS '03)*, pages 131–145, Warszawa, 2003.

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

[7] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, 1992.

[8] F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[9] F. Nielson and H. Seidl. Control-flow analysis in cubic time. In *Proc. ESOP '01*, number 2028 in Lecture Notes in Computer Science, pages 252–268. Springer-Verlag, 2001.

[10] Flemming Nielson, Rene Rydhof Hansen, and Hanne Riis Nielson. Abstract interpretation of mobile ambients. *Science of Computer Programming*, 47:145–175, 2003.

[11] Flemming Nielson, Hanne Riis Nielson, Rene Rydhof Hansen, and Jacob Grydholt Jensen. Validating firewalls in mobile ambients. In *International Conference on Concurrency Theory*, pages 463–477, 1999.

[12] P.Malacaria and C.Hankin. A new approach to control flow analysis. In *Computational Complexity*, pages 95–108, 1998.

[13] P.Malacaria and C.Hankin. Non-deterministic games and program analysis: An application to security. In *Proceedings of the 14th Annual IEEE Symposium on Logic In Computer Science*, pages 443–452. IEEE Computer Society Press, 1999.

[14] Sam B. Sanjabi and Clark Verbrugge. Points-to inspired static analysis for the π-calculus. Sable Technical Report 2003-02, McGill University, Department Of Computer Science, Aug 2003. Available from `http://www.sable.mcgill.ca`.