Type Based Discretionary Access Control*

Michele Bugliesi, Dario Colazzo, and Silvia Crafa

Università Ca' Foscari, Venezia

Abstract. Discretionary Access Control (DAC) systems provide powerful mechanisms for resource management based on the selective distribution of capabilities to selected classes of principals. We study a type-based theory of DAC models for concurrent and distributed systems represented as terms of Cardelli, Ghelli and Gordon's pi calculus with groups [3]. In our theory, groups play the rôle of principals, and the structure of types allows fine-grained mechanisms to be specified to govern the transmission of names, to bound the (iterated) re-transmission of capabilities, to predicate their use on the inability to pass them to third parties, ... and more. The type system relies on subtyping to help achieve a selective distribution of capabilities, based on the groups in control of the communication channels. Type preservation provides the basis for a safety theorem stating that in well-typed processes all names flow according to the delivery policies specified by their types, and are received at the intended sites with the intended capabilities.

1 Introduction

Type systems have been applied widely in process calculi to provide static guarantees for a variety of safety and security properties, including policies for access control [6, 4], non-interference [12, 17, 10], secrecy [1, 3]. In this paper we focus on access control in distributed systems based on Discretionary Access Control (DAC) models [13, 18], i.e. resource management systems that support fine-grained policies for the distribution of capabilities to selected classes of users. To motivate, we start with a well-known example from Pierce and Sangiorgi's seminal paper on types for the pi calculus [16]:

$$S = !s(x).\overline{print}\langle x\rangle \qquad\qquad C = \overline{s}\langle j_1\rangle.\overline{s}\langle j_2\rangle.\dots$$

S is a print spooler serving print requests from a channel s; C is a client sending jobs $j_1, j_2, \dots$ to the spooler. Given this specification, one may wish to show that each of the jobs sent by C is eventually received and printed. While such guarantees can been made for the system $S \mid C$, they may hardly be enforced in more general situations. A misbehaved client C' may participate to the protocol, as in $S \mid C \mid C'$, to steal C's jobs: $C' = s(x).s(y).\dots$. The capability-based type system developed in [16] prevents this unwanted behavior by requiring that all clients be only granted write capabilities on the channel s, and by reserving read capabilities on s to the spooler:

$$(\nu d : ((T)^{\mathsf{rw}})^{\mathsf{rw}})(\nu s : (T)^{\mathsf{rw}})(\,\overline{d}\langle s\rangle \mid \overline{d}\langle s\rangle \;\mid\; d(x : (T)^{\mathsf{r}}).S \;\mid\; d(y : (T)^{\mathsf{w}}).C\,)$$

* Work partially supported by EU-FET project 'MyThS' IST-2001-32617.

The types $(T)^{\nu}$, with $\nu \in \{\mathsf{r}, \mathsf{w}, \mathsf{rw}\}$, indicate the types of channels carrying values of type T with the associated capabilities for reading, writing, or both. By delivering the channel s at different types we can thus enforce an access control policy stating that only the spooler can read jobs.

Notice, however, that the ability of that type system to control the behavior of the system is still rather limited. Indeed, if we want to prevent client jobs from being read by any process other than the spooler S, we need to disallow situations like the following:

$$!s(x).\overline{log}\langle x\rangle.\overline{print}\langle x\rangle \;\mid\; \overline{s}\langle j_1\rangle.\overline{s}\langle j_2\rangle \;\mid\; log(y).SPY$$

where the spooler forwards each of the jobs it receives to process SPY. The capability-based access control from [16] is of little help here, unless one resorts to a more complex encoding of the system or imposes overly restrictive conditions (e.g. prevent the server from writing on all public channels).

A similar problem arises in the following variation of the protocol, in which clients request an ack message to be notified that their jobs have been printed.

$$S = (\nu print)\,(!s(x).\overline{print}\langle x\rangle \mid !print(x).(P \mid \overline{ack}\langle x\rangle)) \qquad C = \overline{s}\langle j_1\rangle.ack(x).\overline{s}\langle j_2\rangle.ack(y)$$

As in the previous case, the capability-based type system will fail to detect violations of the intended protocol due to malicious (or erroneous) servers that discard jobs by, say, running the process $s(x).\overline{ack}\langle x\rangle$.

To counter these problems, we propose a novel typing discipline in which we complement the capability-based control system of [16] with a richer class of types that convey information needed to describe, and prescribe, the ways that values may be exchanged within the different system components. The new types have the form $\mathsf{G}[T \parallel \Delta]$, where G identifies the authority in control of the values of that type, T describes the structure of those values, and Δ is a *delivery policy* governing the circulation of such values along the channels of the system. To illustrate, the typing

$$j : \mathsf{Job}[\;\mathsf{file\_desc} \parallel \mathsf{Spool} \to \mathsf{Print} \to \mathsf{Client}\;]$$

construes j as a file descriptor to be first delivered to the spooler, then passed on to the printer, and only then re-transmitted back to clients for notification. Equivalently, the delivery policy associated with the type of jobs states that (i) no notification should be given to clients for jobs that have not previously been sent to a printer, and that (ii) no job with such type should be received by the printer unless it has first been delivered to the spooler. Similarly, the typing

$$s : \mathsf{Spool}[\;(\mathsf{Job}[\mathsf{file\_desc} \parallel \mathsf{Print} \to \mathsf{Client}])^{\mathsf{rw}} \;\parallel\; -\,]$$

defines s as a (full-fledged) channel, in control of the spooler, and carrying file descriptors which may be passed on to a client only after having been transmitted to the printer (in addition, the type states that s itself should not be re-transmitted). Given the two type assumptions for j and s, our type system will guarantee that transmitting j over s is a well-defined, and legal, operation. Remarkably, this requires a non-standard typing of the output prefix, one that guarantees that j is received by s at the type $\mathsf{J} = \mathsf{Job}[\mathsf{file\_desc} \parallel \mathsf{Print} \to \mathsf{Client}]$, so that j may only be further re-directed to a printer, as expected.

The type system allows for a wide range of delivery policies to be specified, from policies that support delivery chains of unbounded depth, by resorting to recursively defined types, to policies based on multiple, possibly branching, delivery chains along alternative paths, as in $\mathsf{G}[T \parallel \mathsf{G}_1 \to \cdots \to \mathsf{G}_n \,;\, \mathsf{G}'_1 \to (\mathsf{G}'_2 \to \mathsf{G}'_3 \,;\, \mathsf{G}'_4 \to (\mathsf{G}'_5 \,;\, \mathsf{G}'_6))\,]$. To illustrate, in our printer example, the initial typing for the channel s should be defined as follows (with J as given above):

$$s : \mathsf{Spool}[(\mathsf{J})^{\mathsf{rw}} \parallel \mathsf{Spooler}@(\mathsf{J})^{\mathsf{r}};\ \mathsf{Client}@(\mathsf{J})^{\mathsf{w}}].$$

This typing guarantees that s is received at the expected types, namely with read and write capabilities at the spooler and client sites respectively. Relying on similar typing disciplines, one may guarantee that client jobs remain confined within the printer authority, and thus ensure they are not logged and/or leaked to any spy process.

Furthermore, the capability-based access control from [16], based on subtyping, is still available in our system to selectively advertise values at different types depending on the different principals at which they are delivered, as in

$$\mathsf{G}[T \parallel \mathsf{G}_1@T_1 \to \cdots \to \mathsf{G}_n@T_n \ ;\ \mathsf{G}'_1@T'_1 \to (\mathsf{G}'_2@T'_2 \to \mathsf{G}'_3@T'_3 \,;\, \mathsf{G}'_4@T'_4)\,].$$

Remarkably, the types at the intermediate delivery steps may be different, as long as they are all super-types of the type decided at the originating site.

In the rest of the paper we formalize the approach we have outlined in a typed extension of the pi calculus with groups of [3]. We inherit the syntax of the calculus from [3], and introduce a novel operational semantics to express the flow of names during the computation. We also extend the structure of types to capture the access control policies of interest, and we devise a novel typing system for the analysis and the static detection of violations of such policies. The resulting typing discipline is rather flexible and expressive, as we show by providing several examples of powerful discretionary access control policies formalized in our system. A type preservation theorem, proved for the system, allows us to derive a strong safety result stating that all well-typed processes comply with the discretionary access control policies governing the use of resources.

Plan of the paper. §2 reviews the pi calculus with groups from [3], introduces the 'flow' semantics and our new classes of types and illustrates their use with a number of examples. §3 describes the typing system. §4 reports on the properties of the type system, on the relationships with the system in [3]. §5 concludes with final remarks and a discussion of related work.

2 The Typed Calculus

The syntax of processes is the same as in the pi calculus with groups [3] (πG for short) summarized below. We presuppose countable sets of names $n, m, \ldots$, and variables x, y, z, reserving a, b, c to range over both sets. We also presuppose a countable set of *group names* $\mathsf{G}_1, \mathsf{G}_2, \ldots$ containing the distinguished group $\mathsf{Default}$. The notions of free names for a process P, noted $fn(P)$, and free groups in a type T, noted $fg(T)$ are just as in πG.

$$P, Q ::= \mathbf{0} \mid c(\tilde{x}:\tilde{\tau}).P \mid \overline{c}\langle\tilde{a}\rangle.P \mid (\nu n:\tau)P \mid (\nu \mathsf{G})P \mid P|Q \mid !P$$

The novelty with respect to πG is in the structure of types. As in that case, our types are built around group names, but they have a richer structure. Specifically, we interpret groups as representing the authorities (principals) in control of the resources. In addition, drawing inspiration from [15], we structure types so as to convey information on how such resources should be propagated to other principals. The syntax of types is given by the following productions:

$$\begin{array}{rll} \textit{Structural Types} & T ::= \mathsf{B} \mid (\tau_1,\ldots,\tau_n)^{\nu} & (\nu \in \{\mathsf{r},\mathsf{w},\mathsf{rw}\}, \tau_i \textit{ closed}) \\ \textit{Resource Types} & \tau ::= X \mid \mu X.\mathsf{G}[T \parallel \Delta\{X\}] \mid \mathsf{G}[T \parallel \Delta] & \\ \textit{Delivery Policies} & \Delta ::= [\mathsf{G}_i \to \tau_i]_{i\in I} & (\mathsf{G}_i = \mathsf{G}_j \Rightarrow i = j) \end{array}$$

A structural type conveys structural information on the values with that type, i.e. whether they are basic values, of type B, or communication channels: as in other systems, a channel type specifies the types τ_i of the values transmitted over the channel together with the capabilities ν associated with the channel [1]

A resource type is built around a structural type and it additionally specifies the group, or authority, that is in control of the values with that type, together with a set of delivery constraints. A delivery policy, in turn, specifies which other authorities, if any, may legally be granted access to the resource and the extent of the associated access rights (i.e. the capabilities delivered to such authorities). In addition, a delivery policy may impose bounds on the iterated re-distribution of capabilities, and or predicate the delivery of values to the inability to pass them to third parties.

Resource types are recursive, to make it possible to express policies that allow the re-transmission of value to an unbounded depth/distance. Instead, for conceptual simplicity (only), we disallow direct recursion on structural types. We assume type equality up to (*i*) renaming of bound type variables, (*ii*) permutation of delivery constraints inside delivery policies (e.g. $\mathsf{G}[T \parallel [\mathsf{G}_1 \to \tau_1;\ \mathsf{G}_2 \to \tau_2] = \mathsf{G}[T \parallel [\mathsf{G}_2 \to \tau_2;\ \mathsf{G}_1 \to \tau_1])$, and (*iii*) unfolding of recursive types (i.e. $\mu X.\mathsf{G}[T \parallel \Delta\{X\}] = \mathsf{G}[T \parallel \Delta\{\mu X.\mathsf{G}[T \parallel \Delta\{X\}]\}]$). We introduce a number of conventions to ease the notation. We write $\mathsf{G}[T]$ for the type $\mathsf{G}[T \parallel [\,]]$, whose delivery policy is empty, and introduce the following simplified syntax for (finite-depth) delivery chains:

$$\begin{array}{l} \mathsf{G}[T \parallel \mathsf{G}_1@T_1 \to \mathsf{G}_2@T_2 \to \cdots \to \mathsf{G}_n@T_n] \\ \qquad\qquad \triangleq \mathsf{G}[T \parallel [\mathsf{G}_1 \to \mathsf{G}[T_1 \parallel [\mathsf{G}_2 \to \mathsf{G}[T_2 \parallel [\ldots[\mathsf{G}_n \to \mathsf{G}[T_n]]\ldots]]]]]] \end{array}$$

$$\mathsf{G}[T \parallel \mathsf{G}_1 \to \mathsf{G}_2 \to \cdots \to \mathsf{G}_n] \triangleq \mathsf{G}[T \parallel \mathsf{G}_1@T \to \mathsf{G}_2@T \to \cdots \to \mathsf{G}_n@T]$$

2.1 Operational Semantics

The intention of the type system is to control the flow of names, so as to provide static guarantees against any leakage of names to unintended users and/or at unintended types.

[1] In principle, capabilities may be defined to control the access and use of values of any type. We restrict to channel capabilities for presentation purposes only.

Expressing flows is subtle, however, because different occurrences of the same name may flow along different paths during the computation. To illustrate, consider a type $\tau = \mathsf{G}[\mathsf{B} \parallel \mathsf{G}_1 \rightarrow \mathsf{G}_2; \mathsf{G}_3]$ describing values of basic type B to be delivered either to G_1 and then on to G_2, or to G_3 (always at the structural type B). Assume, further, that we are given the following two processes where $n_1{:}\mathsf{G}_1[\dots], n_2{:}\mathsf{G}_2[\dots], n_3{:}\mathsf{G}_3[\dots]$ and $m{:}\tau$:

$$P \triangleq \overline{n_1}\langle m\rangle \mid \overline{n_3}\langle m\rangle \mid n_1(x).n_3(y).\overline{n_2}\langle x\rangle, \quad Q \triangleq \overline{n_1}\langle m\rangle \mid \overline{n_3}\langle m\rangle \mid n_1(x).n_3(y).\overline{n_2}\langle y\rangle$$

Then, P should be judged safe, as both copies of m flow along legal paths, while Q should be deemed unsafe, as it allows an illegal flow for m. However, with the standard reduction semantics these judgments may hardly be made, as after two reduction steps both P and Q reduce to $\overline{n_2}\langle m\rangle$.

To make the notion of flow explicit in the calculus, we resort to a non-standard semantics that uses tags for each name to trace the sequence of channels traversed by the name during the computation. To illustrate, the tagged name $n_{[mpq]}$ represents the name n that flowed first through the channel m, then through p and finally through q. Here 'mpq' is short for the extended notation $m :: p :: q :: \varepsilon$, where ε denotes the empty sequence and '::' is the usual right-associative sequence constructor. We let φ range over sequences of names, and use the notation $\varphi \cdot n$ to indicate the sequence resulting from appending the name n to the tail of φ.

The resulting reduction relation is defined over the class of *dynamic* processes, noted $A, B, \dots$. The structure of dynamic processes coincides with the structure of processes, except that the former may use tagged names in addition to names. Indeed, processes may be understood as special cases of dynamic processes in which all names are tagged with the empty name sequence (e.g. $n_{[\varepsilon]}$, that we identify with n). The notion of free names for dynamic processes is redefined to account for presence of the tags. Specifically, $fn(n_{[\varphi]}(\tilde{x} : \tilde{\tau}).A) = fn(A) \cup \{n\} \cup \{m \mid m \in \varphi\}$, and similarly for the remaining (dynamic) process constructs.

The dynamics of the calculus, in Table 1, is defined as usual in terms of an auxiliary relation of structural congruence. Structural congruence is exactly as in the πG, but relies on the different definition of free names discussed above. The core of the reduction relation is in the (*red comm*) rule, which updates the flow tags to reflect the flow of each of the arguments through the synchronization channel. With this notion of reduction we may now judge process P above safe, and Q unsafe, as the two reduction sequences

$$P \longrightarrow\longrightarrow \overline{n_2}\langle m_{[n_1]}\rangle \quad \text{and} \quad Q \longrightarrow\longrightarrow \overline{n_2}\langle m_{[n_3]}\rangle$$

exhibit the different flow of m in the two computations. We remark here that the tags are only instrumental to record the flow of each name, and have no further effect on the reductions available for processes. We make this precise by relating our flow-sensitive semantics of Table 1 with the original reduction semantics of πG. The latter, which we denote with $\mapsto$, is defined on processes (rather than on dynamic processes) exactly as we do here, but uses the standard communication rule, namely:

$$\overline{n}\langle m_1, \dots, m_k\rangle.P \mid n(x_1{:}\tau_1, \dots, x_k{:}\tau_k).Q \;\mapsto\; P \mid Q\{{}^{m_i}/_{x_i}\}$$

in place of our (*red comm*) rule. Given any dynamic process A, let $|A|$ denote the (proper) process resulting from erasing all the tags from A. Then we have:

Table 1 The operational semantics

Dynamic Processes: $a,b,\ldots$ denote either variables of tagged names of the form $n_{[\varphi]}$.

$$A ::= \mathbf{0} \mid a(x_1{:}\tau_1,\ldots,x_n{:}\tau_n).A \mid \overline{a}\langle b_1,\ldots,b_n\rangle.A \mid (\nu n:\tau)A \mid (\nu \mathsf{G})A \mid A|B \mid !B$$

Structural congruence

(*monoid*))	$A \mid B \equiv B \mid A, \quad A \mid \mathbf{0} \equiv A, \quad (A \mid B) \mid C \equiv A \mid (B \mid C)$	
(*repl*))	$!A \equiv A \mid !A$	
(*name extr*)	$(\nu n:\tau)(A \mid B) \equiv A \mid (\nu n:\tau)B$	if $n \notin fn(A)$
(*group extr*)	$(\nu \mathsf{G})(A \mid B) \equiv A \mid (\nu \mathsf{G})B$	if $\mathsf{G} \notin fg(A)$
(*name/group exch*)	$(\nu \mathsf{G})(\nu n:\tau)A \equiv (\nu n:\tau)(\nu \mathsf{G})A$	if $\mathsf{G} \notin fg(\tau)$
(*name exch*)	$(\nu n_1:\tau_1)(\nu n_2:\tau_2)A \equiv (\nu n_2:\tau_2)(\nu n_1:\tau_1)A$	
(*group exch*)	$(\nu \mathsf{G}_1)(\nu \mathsf{G}_2)A \equiv (\nu \mathsf{G}_2)(\nu \mathsf{G}_1)A$	

Reduction

(*red comm*)	$\overline{n_{[\varphi]}}\langle m_{1[\varphi_1]},\ldots,m_{k[\varphi_k]}\rangle.A \mid n_{[\psi]}(x_1{:}\tau_1,\ldots,x_k{:}\tau_k).B \longrightarrow A \mid B\{^{m_{i[\varphi_i\cdot n]}}/_{x_i}\}$
(*red res*)	$A \longrightarrow A' \implies (\nu n{:}\tau)A \longrightarrow (\nu n{:}\tau)A'$
(*red group*)	$A \longrightarrow A' \implies (\nu \mathsf{G})A \longrightarrow (\nu \mathsf{G})A'$
(*red par*)	$A \longrightarrow A' \implies A \mid B \longrightarrow A' \mid B$
(*red struct*)	$A \equiv\longrightarrow\equiv B \implies A \longrightarrow B$

Theorem 1 (Flow reduction vs Standard reduction). *Let A be a closed dynamic process. If $A \longrightarrow^* B$ then $|A| \mapsto^* |B|$. Conversely, if $|A| \mapsto^* Q$ then there exists B such that $A \longrightarrow^* B$ and $|B| \equiv Q$.*

This result would hold also in the presence of a matching operator to compare names. In particular, similarly to our (*red comm*) rule, in dynamic processes the matching construct would disregard the tags to decide name equality, as in the following reductions:

$$\textsf{if } [n_{[\varphi_1]} = n_{[\varphi_2]}] \textsf{ then } P \textsf{ else } Q \longrightarrow P$$
$$\textsf{if } [n_{[\varphi_1]} = m_{[\varphi_2]}] \textsf{ then } P \textsf{ else } Q \longrightarrow Q \quad (n \neq m)$$

Besides being coherent with our current development, disregarding the tags to test name equality is crucial to encode any sensible form of matching. The simplest illustration is probably the case of nonce-based authentication protocols, in which a principal generates a nonce (i.e. a fresh name) and then uses matching to test the freshness of an incoming message by comparing the nonce included in the message with the name it generated: clearly, this test may only succeed if we disregard the nonce's flow.

2.2 Types and Discretionary Access Control Policies

To start illustrate of our types, let pwd be a basic type describing passwords. Then we may define the type $\mu X.\mathsf{G}[\mathsf{pwd} \parallel \mathsf{G} \to X]$ to constrain the free re-transmission of values with this type only within members of group G. Alternatively, we may define the type $\mu X.\mathsf{G}[\mathsf{pwd} \parallel \mathsf{G} \to X \ ; \ \mathsf{F} \to \mathsf{G}[\mathsf{pwd}]]$ to qualify passwords that may also be passed to friends, of group F, provided that they do not pass them over to third parties.

The re-transmission of values may also be filtered on the basis of the capabilities that are passed along with the values. For instance, given $\tau = \mu Y.\mathsf{G}[(\mathsf{nat})^{\mathsf{w}} \parallel \mathsf{F} \to Y]$, we may define the type $\mu X.\mathsf{G}[(\mathsf{nat})^{\mathsf{rw}} \parallel \mathsf{G} \to X;\ \mathsf{F} \to \tau]$ to qualify nat channels of group G that may be received and re-transmitted at group F at a type that restricts their use as to write-only channels[2]. Similarly, the type

$$\mu X.\mathsf{G}[(\mathsf{nat})^{\mathsf{rw}} \parallel [\ \mathsf{G}_1 \to \mathsf{G}[(\mathsf{nat})^{\mathsf{w}} \parallel \Delta_1]\ ; \quad \mathsf{G}_2 \to \mathsf{G}[(\mathsf{nat})^{\mathsf{r}} \parallel \Delta_2]\ ; \quad \mathsf{Default} \to X]\ ]$$

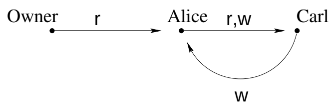
qualifies nat channels owned by G, that can be delivered to group G_1 and G_2 as write-only and read-only channels respectively. Instead, no restriction applies for other groups, as indicated by the $\mathsf{Default}$ entry in the delivery policy. The type also indicates the delivery policies Δ_1 and Δ_2, that we leave unspecified here, for the subsequent 'hops', from G_1 and G_2 respectively. We further discuss the import of resource types in formalizing examples of DAC policies below.

Most forms of discretionary access control focus on the so called *owner-based* administration of access rights, by which the owner of an object, typically its originator, has the discretionary authority over who else can access that object. DAC policies vary depending on how the owner's discretionary power can be delegated to other users.

In *strict DAC* ([18, 19]), the owner is the only entity to have the authority to grant access to an object. Such policies are directly expressed in our type system with types with a rather regular structure, namely $\mathsf{Owner}[T \parallel [\mathsf{User_i} \to \tau_i]_{i \in I}]$ where τ_i is the resource type that constrains the re-transmission of its values only within the authority of $\mathsf{User_i}$, namely: $\tau_i = \mu X.\mathsf{Owner}[T \parallel \mathsf{User_i} \to X]$.

Liberal DAC ([18, 19]), models are more flexible, and interesting. They are based on a decentralized authorization policy where the owner of an object delegates other users the privilege of specifying authorizations, possibly with the ability of further delegating it. Two popular classes of Liberal DAC policies are those known as *originator controlled* [14], and *true delegation* [18].

An *originator controlled* policy prevents access to data being extended to any authority without the owner's explicit permission. In this case when a resource is created the owner has full control over how the resource's capabilities can be distributed. An example of such policies is given the diagram below, representing an Owner that creates a channel and specifies how it should be distributed to two parties, Alice and Carl:



The channel is first delivered, in read mode, to Alice, who is delegated to re-transmit it to Carl with the additional write capability; only then is Alice allowed to receive the write capability from Carl. This delivery policy, imposed by the owner to ensure that

[2] Here, and elsewhere, we say *transmit at a group* to mean *transmit at channels of that group*.

Alice will not write on the new channel until it has been received also by Carl, may be expressed with our types as follows:

$$\mathsf{Owner}[(T)^{\mathsf{rw}} \parallel \mathsf{Alice}@(T)^{\mathsf{r}} \;\rightarrow\; \mathsf{Carl}@(T)^{\mathsf{rw}} \;\rightarrow\; \mathsf{Alice}@(T)^{\mathsf{w}}]$$

A similar example can be recovered from the literature on cryptographic protocols. Consider the case where two parties, Alice and Bob, wish to establish a private exchange. To accomplish that, Alice creates a fresh name, say c_{AB}, sends it to a trusted Server and delegates it to forward the name to Bob so that the exchange may take place. Here, the Server should only act as a forwarder, and not interfere with the exchanges between Alice and Bob. This can be achieved using the typing

$$c_{AB} : \mathsf{Alice}[(\mathsf{data})^{\mathsf{rw}} \parallel \mathsf{Server}@(\mathsf{data}) \rightarrow \mathsf{Bob}@(\mathsf{data})^{\mathsf{rw}}]$$

in which the the Server receives the channel with no capability, as intended.

True delegation policies are more liberal, and allow any principal receiving the grant option to further distribute it to principals not explicitly anticipated by the owner. Such policies are formalized in our system with the help of Default entries. To illustrate, a policy in which the owner principal gives full discretionary power to a delegate, may be expressed by the type $\mathsf{Owner}[T \parallel \mathsf{D} \rightarrow \tau]$, where $\tau = \mu X.\mathsf{Owner}[T \parallel \mathsf{Default} \rightarrow X]$.

3 Typing and Subtyping

Having illustrated how types may be employed to formalize discretionary policies, we now turn to the problem of analyzing the import of such policies on the dynamic behavior of processes. Notice, to this regard, that the delivery policies we outlined in the previous section rely critically on the ability to deliver names at types that may vary non-monotonically along the delivery chains. Then, to have the desired safety guarantees, we must envision a mechanism to ensure that the types at each delivery site be a safe approximation of the type at the originating site. This is accomplished by the rules for type formation and subtyping we discuss next.

3.1 Type formation and subtyping

Type formation is defined in Table 2. Any type must be built around group names and type variables known to the type environment. In addition any resource type, say $\mathsf{G}[T \parallel \Delta]$, must be so defined as to ensure that each resource type occurring in the delivery policy Δ is built around (i) the group name G of the owner, and (ii) a structural super-type of the structural type T. The first constraint could be lifted, but we do not explore this possibility here; the second constraint, instead, is critical for soundness as we just observed. The type formation rules enforce both constraints with the help of the auxiliary judgments $\Gamma \vdash_{\mathsf{G},T} \tau$, that verify τ in relation to the group G and the type T, for all the resource types τ introduced by Δ.

The subtyping relation is axiomatized in Table 3. The first two blocks of rules are standard (cf. [16]). The core of the subtyping relation is in the last two rules. Rule

Table 2 Type Formation

Resource Types – owner

$$(\tau\text{-OWNER})\ \frac{\Gamma \vdash_{\mathsf{G},T} \tau_i \quad fg(\mathsf{G}[T \parallel [\mathsf{G}_i \to \tau_i]]) \subseteq Dom(\Gamma)}{\Gamma \vdash \mathsf{G}[T \parallel [\mathsf{G}_i \to \tau_i]_{i\in I}]} \qquad (\tau\text{-VAR})\ \frac{X \in \Gamma \quad \Gamma \vdash \diamond}{\Gamma \vdash X} \qquad (\tau\text{-REC})\ \frac{\Gamma, X \vdash \mathsf{G}[T \parallel \Delta\{X\}]}{\Gamma \vdash \mu X.\mathsf{G}[T \parallel \Delta\{X\}]}$$

Resource Types – step

$$(\tau\text{-STEP})\ \frac{\Gamma \vdash T \leq T' \quad \Gamma \vdash_{\mathsf{G},T} \tau_i}{\Gamma \vdash_{\mathsf{G},T} \mathsf{G}[T' \parallel [\mathsf{G}_i \to \tau_i]_{i\in I}]} \qquad (\tau\text{-VAR})\ \frac{X \in \Gamma \quad \Gamma \vdash \diamond}{\Gamma \vdash_{\mathsf{G},T} X} \qquad (\tau\text{-REC})\ \frac{\Gamma, X \vdash_{\mathsf{G},T'} \mathsf{G}[T \parallel \Delta\{X\}]}{\Gamma \vdash_{\mathsf{G},T'} \mu X.\mathsf{G}[T \parallel \Delta\{X\}]}$$

(τ-TYPE) makes resource-subtyping covariant in the component structural types, as expected, and required for soundness. More interestingly, (τ-POLICY) requires resource types to impose more restrictive delivery policies than their sub-types. At a first look, the ordering relation on delivery policies is reminiscent of the subtype relation on record types. This is indeed the case if we restrict to delivery policies without Default entries and predicate $\Delta \preccurlyeq \Delta'$ to the additional constraint that $Dom(\Delta') \subseteq Dom(\Delta)$. The resulting $\preccurlyeq$ relation captures a form of originator controlled DAC model, in which resource owners have full delivery control over their resources. On the other hand, in case $Dom(\Delta') \supset Dom(\Delta)$, and similarly with policies that include Default entries, the ordering relation on policies captures DAC models with *true delegation*, in which intermediate users of a resource may autonomously make decisions on the next delivery steps, provided that they advertise the resource at structural super-type of the owner's structural type.

To illustrate, consider first $\Delta = [\mathsf{Default} \to \mathsf{G}[\tau^{\mathsf{rw}}]]$ and $\Delta' = [\mathsf{G}_1 \to \mathsf{G}[\tau^{\mathsf{r}}];\ \mathsf{G}_2 \to \mathsf{G}[\tau^{\mathsf{w}}]]$. For a proper Γ one has $\Gamma \vdash \Delta \preccurlyeq \Delta'$, which provides support for true delegation as Δ allows any delivery, while Δ' distributes the read capability to channels of group G_1 and the write capability to channels of group G_2. As a further example, consider $\Delta = [\mathsf{G}_1 \to \mathsf{G}[\tau^{\mathsf{r}}];\ \mathsf{Default} \to \mathsf{G}[\tau^{\mathsf{rw}}]]$ and $\Delta' = [\mathsf{Default} \to \mathsf{G}[\tau^{\mathsf{rw}}]]$. Here Δ allows all groups but G_1 to receive the write capability, whereas Δ' does not make this distinction. The two policies are not related by the $\preccurlyeq$ relation, as $\Gamma \vdash \Delta \preccurlyeq \Delta'$ requires that in the presence of Default entries in both Δ and Δ', all delivery constraint expressed in Δ must also be enforced by Δ'.

3.2 Typing of processes and dynamic processes

The typing rules for (dynamic) processes, in Table 4, complete the presentation of the type system. We remark that the typing rules validate dynamic processes, hence also the (proper) processes of the source calculus. This is required for subject reduction, as the result of a reduction (sequence) is a dynamic process rather than a process.

The typing rules for names allow each name to be typed at (any super-type of) the resource type τ known to the environment, by rules (PROJECT) and (SUBSUMPTION),

Table 3 Subtypes

(REFLEX)
$$\frac{\Gamma \vdash \diamond \quad fg(\tau) \subseteq Dom(\Gamma)}{\Gamma \vdash \tau \leq \tau}$$

(TRANS)
$$\frac{\Gamma \vdash \tau \leq \rho \quad \Gamma \vdash \rho \leq \sigma}{\Gamma \vdash \tau \leq \sigma}$$

Structural subtypes:

(T-WRITE)
$$\frac{\Gamma \vdash \tau'_1 \leq \tau_1 \quad \cdots \quad \Gamma \vdash \tau'_n \leq \tau_n \quad n \geq 1}{\Gamma \vdash (\tau_1, \ldots, \tau_n)^{\mathsf{w}} \ \leq \ (\tau'_1, \ldots, \tau'_n)^{\mathsf{w}}}$$

(T-READ)
$$\frac{\Gamma \vdash \tau_1 \leq \tau'_1 \quad \cdots \quad \Gamma \vdash \tau_n \leq \tau'_n \quad n \geq 1}{\Gamma \vdash (\tau_1, \ldots, \tau_n)^{\mathsf{r}} \ \leq \ (\tau'_1, \ldots, \tau'_n)^{\mathsf{r}}}$$

(T-READ/WRITE)
$$\frac{\Gamma \vdash \diamond \quad n \geq 0}{\Gamma \vdash (\tau_1, \ldots, \tau_n)^{\mathsf{rw}} \ \leq \ (\tau_1, \ldots, \tau_n)^{\mathsf{v}}}$$

Resource Subtypes

Let $\Gamma \vdash \Delta \preccurlyeq \Delta' \triangleq$

$\forall \mathsf{G} \in Dom(\Delta')$ if $(\mathsf{G} \in Dom(\Delta))$ then $\Gamma \vdash \Delta(\mathsf{G}) \leq \Delta'(\mathsf{G})$ else $\Gamma \vdash \Delta(\mathsf{Default}) \leq \Delta'(\mathsf{G})$
and if $\mathsf{Default} \in Dom(\Delta) \cap Dom(\Delta')$ then $\forall \mathsf{G} \in Dom(\Delta) \setminus Dom(\Delta').\Gamma \vdash \Delta(\mathsf{G}) \leq \Delta'(\mathsf{Default})$

(τ-TYPE)
$$\frac{\Gamma \vdash T \leq T' \quad fg(\mathsf{G}[T \parallel \Delta]) \cup fg(T') \subseteq Dom(\Gamma)}{\Gamma \vdash \mathsf{G}[T \parallel \Delta] \ \leq \ \mathsf{G}[T' \parallel \Delta]}$$

(τ-POLICY)
$$\frac{\Gamma \vdash \Delta \preccurlyeq \Delta' \quad fg(\mathsf{G}[T \parallel \Delta]) \cup fg(\Delta') \subseteq Dom(\Gamma)}{\Gamma \vdash \mathsf{G}[T \parallel \Delta] \ \leq \ \mathsf{G}[T \parallel \Delta']}$$

as well as at any type mentioned at the subsequent hops of each chain of the delivery policy associated with τ. One application of rule (DELIVERY) reaches the types at the first hops; subsequent applications reach types occurring further on along the chains.

This ability to type names at all their delivery types is crucial to the proof of subject reduction. To see why, we first look at rule (OUTPUT), the core of the delivery discipline. Consider the case when name, say m is emitted on a channel, say $n : \mathsf{G}[(\tau)^{\mathsf{w}}]$. Assume further that m is known to the environment at type $\mathsf{F}[T \parallel \Delta]$. Rule (OUTPUT) verifies that G is indeed one of the next 'hops' in Δ, and the type τ' at which m should be delivered to G is a subtype of τ, the type of values carried by n. Given that the types of names known to the environment must be well-formed, the type formation rules ensure that the original structural type of m is a subtype of the structural component of τ', thus also a subtype of the type τ at which n expects to receive its values: this guarantees that m may safely be received at n. Further re-transmissions of m will undergo the same checks by the (OUTPUT) rule, but now with the types advertised at the subsequent hops in Δ: to prove subject reduction we therefore need to be able to type m at all such types. Except for this specificity in the typing of names, the proof of subject reduction for the system is standard (cf. §4).

We give an example to show the effect of value propagation with a typed version of the print spooler discussed in the introduction. Let $!s(x{:}\tau_1).\overline{print}\langle x\rangle \mid !print(x : \tau_2).(P \mid \overline{ack}\langle x\rangle) \mid (\nu\, j{:}\tau_J)\ \overline{s}\langle j\rangle.ack(x{:}\tau_3).C$ be a system where a client creates a new job j, sends it to the printer and waits for notification. The desired delivery policy for j requires the job to be sent first to the print spooler, then to the printer, and finally back to the client. Such policy is enforced with the types:

$$\tau_J = \mathsf{Job}[\mathsf{file\_desc} \parallel \mathsf{Spool} \to \tau_1] \qquad \tau_1 = \mathsf{Job}[\mathsf{file\_desc} \parallel \mathsf{Print} \to \tau_2]$$
$$\tau_2 = \mathsf{Job}[\mathsf{file\_desc} \parallel \mathsf{Client} \to \tau_3] \qquad \tau_3 = \mathsf{Job}[\mathsf{file\_desc} \parallel \Delta]$$

and assuming $s : \mathsf{Spool}[(\tau_1)^{\mathsf{rw}}]$ $print : \mathsf{Print}[(\tau_2)^{\mathsf{rw}}]$, and $ack : \mathsf{Client}[(\tau_3)^{\mathsf{rw}}]$. We leave it to the reader to verify that all types involved are well-formed and that the process typechecks.

4 Type System Properties

We conclude the presentation of the type system by elucidating its main properties. To state such properties, we first introduce two additional (partial) operators on types and type environments. Given a type environment Γ and a sequence of names φ we let $\Gamma[\varphi]$ denote the sequence of groups that Γ associates with the names in φ.

$$\begin{aligned} \Gamma[\varepsilon] &\triangleq \varepsilon \\ \Gamma[n :: \varphi] &\triangleq \text{if } (\Gamma(n) = \mathsf{G}[T \parallel \Delta] \text{ and } \Gamma[\varphi] \neq \bot) \text{ then } \mathsf{G} :: \Gamma[\varphi] \text{ else } \bot \end{aligned}$$

Given a closed resource type τ, and a sequence of group names π, we then denote with $\tau \downarrow \pi$, the type occurring in τ at π, in the following sense:

$$\begin{aligned} \tau \downarrow \varepsilon &\triangleq \tau \\ \mathsf{G}[T \parallel \Delta] \downarrow \mathsf{G}' :: \pi &\triangleq \text{if } (\mathsf{G}' \in Dom(\Delta)) \text{ then } \Delta(\mathsf{G}') \downarrow \pi \\ &\quad\ \text{else if } (\mathsf{Default} \in Dom(\Delta)) \text{ then } \Delta(\mathsf{Default}) \downarrow \pi \text{ else } \bot \end{aligned}$$

The next three theorems express the safety properties of the type system Rather than defining an explicit notion of error and showing that well-typed terms don't have error transitions, as in, e.g. [6], we state our safety properties directly in terms of types: the two approaches are essentially equivalent. Theorem 2 states that in all well-typed dynamic processes, any access to a channel complies with the access rights associated with the channel, and the arguments are passed over the channel at subtypes of the expected types. Theorem 3 states that in all well-typed dynamic processes names flow according to the delivery policies expressed by their types. Finally, by Theorem 4 we know that such properties are preserved by reduction. Collectively, these theorems provide static guarantees that in well-typed (proper) processes, all resources are accessed and delivered according to the policies defined by their types.

Theorem 2 (Access Control). *Let A and B be closed, dynamic processes with $A \equiv \overline{n_{[\varphi]}}\langle a_1,\dots,a_k\rangle.A'$ and $B \equiv n_{[\varphi']}(x_1{:}\rho_1,\dots,x_l{:}\rho_l).B'$. Assume, further, that $\Gamma \vdash A \mid B$. Then $l = k$ and one has:*

Table 4 Typing of dynamic processes. (note: we identify $n_{[\varepsilon]}$ with n)

$$\text{(EMPTY)}\ \frac{}{\varnothing \vdash \diamond} \qquad \text{(ENV } n\text{)}\ \frac{\Gamma \vdash \diamond \quad \Gamma \vdash \tau \quad n \notin Dom(\Gamma)}{\Gamma, n : \tau \vdash \diamond} \qquad \text{(ENV } x\text{)}\ \frac{\Gamma \vdash \diamond \quad fg(\tau) \subseteq Dom(\Gamma) \quad x \notin Dom(\Gamma)}{\Gamma, x : \tau \vdash \diamond}$$

$$\text{(ENV G)}\ \frac{\Gamma \vdash \diamond \quad \mathsf{G} \notin Dom(\Gamma)}{\Gamma, \mathsf{G} \vdash \diamond} \qquad \text{(ENV } X\text{)}\ \frac{\Gamma \vdash \diamond \quad X \notin Dom(\Gamma)}{\Gamma, X \vdash \diamond} \qquad \text{(PROJECT)}\ \frac{\Gamma, a : \tau \vdash \diamond}{\Gamma, a : \tau \vdash a : \tau} \qquad \text{(SUBSUMPTION)}\ \frac{\Gamma \vdash a : \tau' \quad \Gamma \vdash \tau' \leq \tau}{\Gamma \vdash a : \tau}$$

$$\text{(DELIVERY)}\ \frac{\Gamma \vdash n_{[\varphi]} : \mathsf{G}[T \parallel \Delta] \quad \Gamma \vdash m : \mathsf{G}_1[T_1 \parallel \Delta_1] \quad (\mathsf{G}_1 \to \tau \in \Delta) \vee (\mathsf{G}_1 \notin Dom(\Delta) \wedge \mathsf{Default} \to \tau \in \Delta)}{\Gamma \vdash n_{[\varphi m]} : \tau}$$

$$\textit{Let } \Gamma \vdash \Delta_i(\mathsf{G}) \preccurlyeq \tau_i \ \triangleq\ \begin{array}{l} \text{if } (\mathsf{G} \in Dom(\Delta_i)) \text{ then } \Gamma \vdash \Delta_i(\mathsf{G}) \leq \tau_i \\ \text{else } (\mathsf{Default} \in Dom(\Delta_i)) \text{ and } \Gamma \vdash \Delta_i(\mathsf{Default}) \leq \tau_i \end{array}$$

$$\text{(INPUT)}\ \frac{\Gamma \vdash a : \mathsf{G}[(\tau_1, \ldots, \tau_k)^{\mathsf{r}} \parallel \Delta] \quad \Gamma, x_1 : \tau_1, \ldots, x_k : \tau_k \vdash A \quad 0 \leq k}{\Gamma \vdash a(x_1 : \tau_1, \ldots, x_k : \tau_k).A}$$

$$\text{(OUTPUT)}\ \frac{\Gamma \vdash a : \mathsf{G}[(\tau_1, \ldots, \tau_k)^{\mathsf{w}} \parallel \Delta] \quad \Gamma \vdash A \quad \Gamma \vdash b_i : \mathsf{G}_i[T_i \parallel \Delta_i] \quad \Gamma \vdash \Delta_i(\mathsf{G}) \preccurlyeq \tau_i \quad 0 \leq i \leq k}{\Gamma \vdash \overline{a}\langle b_1, \ldots, b_k \rangle.A}$$

$$\text{(NEW } G\text{)}\ \frac{\Gamma, \mathsf{G} \vdash A}{\Gamma \vdash (\nu \mathsf{G})A} \qquad \text{(NEW } n\text{)}\ \frac{\Gamma, n : \tau \vdash A}{\Gamma \vdash (\nu n : \tau)A} \qquad \text{(PAR)}\ \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \mid B} \qquad \text{(DEAD)}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0}} \qquad \text{(REPL)}\ \frac{\Gamma \vdash A}{\Gamma \vdash !A}$$

- $\Gamma \vdash n_{[\varphi]} : \mathsf{G}[(\tau_1, \ldots, \tau_k)^{\mathsf{w}}]$, $\Gamma \vdash n_{[\varphi']} : \mathsf{G}[(\rho_1, \ldots, \rho_k)^{\mathsf{r}}]$, *and*
- *for all* $i = 1, ..., k$, $\Gamma \vdash a_i : \sigma_i$ *with* $\sigma_i \downarrow \mathsf{G} \neq \perp$, $\Gamma \vdash \sigma_i \downarrow \mathsf{G} \leq \tau_i$ *and* $\Gamma \vdash \tau_i \leq \rho_i$.

Theorem 3 (Flow Control). *Let* $\Gamma \vdash A$ *with* A *closed. Assume, further, that* $\Gamma \vdash A$ *depends on the judgment* $\Gamma' \vdash n_{[\varphi]} : \tau$. *Then* $n \in Dom(\Gamma')$ *and* $\Gamma'[\varphi] \neq \perp$. *In addition,* $\Gamma'(n) = \rho$ *with* ρ *such that* $\rho \downarrow \Gamma'[\varphi] \neq \perp$ *and* $\Gamma' \vdash \rho \downarrow \Gamma'[\varphi] \leq \tau$.

Theorem 4 (Subject Reduction). *If* $\Gamma \vdash A$ *and* $A \longrightarrow^* B$, *then* $\Gamma \vdash B$.

We remark that theorem 3 could not be meaningfully stated without appealing to the flow tags attached to names. In particular, it is not true, in general, that given an extended process A, $\Gamma \vdash A$ implies $\Gamma \vdash |A|$. To see that, note that $\Gamma \vdash A$ may depend on two tagged names $n_{[\varphi_1]}$ and $n_{[\varphi_2]}$ being given different types, (not related by subtyping) by resorting to the (DELIVERY) rule. By erasing the tags, we lose the possibility of appealing to the (DELIVERY) rule, and consequently the judgement $\Gamma \vdash |A|$ fails. For this very reason, Theorem 4 does not hold, in general, under the reduction semantics $\mapsto$ of [3]. Interestingly, however, we can recover subject reduction for $\mapsto$ provided that we make adequate assumptions on the structure of the types occurring in Γ and A. We formalize the relationship with the type system of [3] below.

4.1 Encoding the pi-calculus with groups

As we mentioned, the syntax of the pi-calculus with groups is the same as the one in §2. The types, instead, are defined simply as follows: $T ::= \mathsf{G}[T_1, \ldots, T_n]$. These types may be encoded into our resource types as follows:

$$[\![\mathsf{G}[T_1, \ldots, T_n]]\!] = \mu X.\mathsf{G}[([\![T_1]\!], \ldots, [\![T_n]\!])^{\mathsf{rw}} \parallel \mathsf{Default} \to X]$$

The encoding provides all types with the most liberal delivery policy, one that allows unboundedly-deep delivery over all channels: the only constraint is that the receiving channels have access to the group of the value they carry with them, exactly as in πG.

We show that our type system is a conservative extension of the type system of πG. Given a πG type environment Γ and a πG process P, let $[\![\Gamma]\!]$ and $[\![P]\!]$ be the type environment and process that result from applying the encoding of types systematically to all types occurring in Γ and P. Then we have:

Theorem 5 (Relationships with πG). *$\Gamma \vdash P$ is derivable in πG iff $[\![\Gamma]\!] \vdash [\![P]\!]$ is derivable in our type system.*

This follows by observing that if we restrict to simple types, the type formation rules as well as the (OUPUT) rule for processes coincide with the corresponding rules in πG. Now, call a type *simple* when it is the encoding of a πG type. Similarly, call a type environment and a (dynamic) process *simple* when all the types occurring therein are simple. If Γ and A are simple, it is not difficult to see that $\Gamma \vdash A$ implies $\Gamma \vdash |A|$. Intuitively, the reason is that simple types are insensitive to flows: this is a consequence of the delivery type being the same at all hops in a simple type. More interestingly, for simple processes we have subject reduction, as we state next.

Theorem 6 (Subject Reduction for simple processes). *Assume $\Gamma \vdash P$ with Γ simple, and P closed and simple, and let $P \mapsto^* Q$. Then $\Gamma \vdash Q$.*

Based on this result, the secrecy theorem of [3] can be re-established in our system with no additional effort for simple processes.

5 Conclusion

We have developed a type theory for the specification and the static analysis of access control policies in the pi calculus. Our approach extends and complements previous work on the subject by introducing a new class of types so defined as to control the dynamic flow of values among system components. We have shown the flexibility of our system with several examples, and proved that it provides strong safety guarantees for all well-typed processes.

There are several desirable extensions to the present system: they include the ability to express the revocation of capabilities, to change the ownership on resources, to account for hierarchical relationships among principals, or for declassification mechanisms. A further topic of investigation is the study of typed equivalences to gain deeper insight into the import of our access control policies in the behavioural properties of processes. We leave all these to our plans for future work, and conclude here with a brief discussion on related work.

While we are not aware of any approach specifically targeted at the access control mechanisms we have discussed here, our work is clearly related to a large body of literature on (type-based) security in process calculi.

Several type system encompass various forms of access control policies in distributed systems. Among them, [4] proposes a form of distributed access control based on typed cryptographic operations; the work on Dπ has produced fairly sophisticated type systems [7, 8] to control the access to the resources advertised at the different locations of a distributed system. None of these systems, however, addresses the kind of discretionary policies we have considered here. More precisely, in Dπ, resources are created at a specific, unique, type and then delivered to different parties at different (super)types on different channels. Unlike in our system, however, the 'delivery policy' of a value is not described (nor prescribed) by its type. As a consequence, the only guarantee offered by the the type system is that names delivered at type T will be received, and re-transmitted freely, at super-types of T. As we have illustrated, our types may be employed to specify, and enforce, much more expressive policies.

Type systems have also been proposed to control implicit information flows determined by the behaviour of system components (see [12, 17, 10, 21] among others). These type systems trace the causality relations between computational steps in order to detect covert channels. We follow a different approach to express and verify the delivery of (and the access to) the system resources.

Our approach is also related to the large body of existing work on security automata. In fact, the delivery policies we express in our type system could equivalently be described as finite-state automata whose states are structural types and edges are labelled with groups. On the other hand, security automata have traditionally been employed to provide for run-time system monitoring [20] rather than as a basis for the development of static, type-based, security analyses. More interestingly, the possibility to structure our types as automata (and similarly, as XML-like regular expressions) should lead to the development of efficient algorithms for (sub-) type checking. We leave this as a subject of future work.

The use of session types [9] to express and gain control over the different steps of communication protocols shares motivations with our approach. Session types provide mechanisms to regulate the sequence of events occurring in a two-parties interaction session. Specifically, a session type prescribes what type of values may legally be passed on a given channel, and in which order. On the other hand, our delivery types prescribe how values can be exchanges among different principals. In both approaches, the types employed are finite-state automata: however the information encoded by the states is different and, as we just argued, largely complementary.

References

1. M. Abadi and A. D. Gordon. Reasoning about cryptographic protocols in the Spi calculus. *Lecture Notes in Computer Science*, 1243, 1997.
2. V. Benzaken, G. Castagna, and A. Frisch. CDuce: a white paper. In *Programming Language Technologies for XML*, Pittsburgh, October 2002. Available at www.cduce.org.
3. L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. In *Proceedings of CONCUR*, number 1877 in Lecture Notes in Computer Science, pages 365–379. Springer-Verlag, 2000.
4. T. Chothia, D. Duggan, and J. Vitek. Type-based distributed access control. In *CSFW 2003*, pages 170–184. IEEE, 2003.
5. Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic Subtyping. In *Proceedings, Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.
6. M. Hennessy and J. Riely. Information flow vs resource access in the asynchronous π-calculus. *ACM Transactions on Programming Languages and Systems*, 24(5):566–591, 2002.
7. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:82–120, 2002.
8. Matthew Hennessy, Julian Rathke, and Nobuko Yoshida. SafeDpi: a language for controlling mobile code. In *FoSSaCS'04, Seventh Int. Conference on Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, 2004. To appear. Also available as Sussex Technical Report cs02:2003.
9. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP '98*, Lecture Notes in Computer Science, pages 122–138. Springer-Verlag, 1998.
10. K. Honda, V.T. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *ESOP '00*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer-Verlag, 2000.
11. Haruo Hosoya. Regular expression types for XML. In *Ph.D. thesis. The University of Tokyo*, 2002.
12. Naoki Kobayashi. Type-based information flow analysis for the pi-calculus. Technical Report TR03-0007, Department of Computer Science, Tokyo Institute of Technology, October 2003.
13. B.W. Lampson. Protection. *ACM Operating Systems Rev.*, 8(1):18–24, Jan. 1974.
14. C.J. McCollum, J. R. Messing, and L. Notargiacomo. Beyond the pale of mac and dac – defining new forms of access control. In *Proc. of IEEE Symposium on Security and Privacy*, pages 190–200, 1990.
15. Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol*, (4):410–442, 2000.
16. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5), 1996.
17. Franois Pottier. A simple view of type-secure information flow in the π-calculus. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 320–330, 2002.
18. P. Samarati and S. De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, LNCS 2171. Springer-Verlag, 2001.

19. Ravi S. Sandhu and Qamar Munawer. How to do discretionary access control using roles. In *ACM Workshop on Role-Based Access Control*, pages 47–54, 1998.
20. Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.
21. P. Sewell and J. Vitek. Secure composition of untrusted code: Boxmpi, wrappers and causality types. *Journal of Computer Security*, 11(2):135–188, 2003.