Combining partitions in SecSpaces

Mario Bravetti, Roberto Gorrieri,
Roberto Lucchi and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura Anteo Zamboni 7, I-40127 Bologna, Italy.
E-mail: {*bravetti, gorrieri, lucchi, zavattar*}*@cs.unibo.it*

Abstract

SecSpaces is a data-driven coordination model that supports a tuple-based coordination space extended with mechanisms for controlling and authenticating the access to entries. This is achieved exploiting the notion of (symmetric and asymmetric) abstract partitions inside the space. In this paper we consider one of the limitations of SecSpaces: it is not well suited for supporting the atomic access to more than one partition at once. In order to tackle this limitation we extend the SecSpaces model introducing an operator to combine partitions; output operations can use the new operator to make an entry visible in more than one partition, and data-retrieval operations can use it to access atomically more than one partition. We formally define, in terms of a process calculus, this notion and we demonstrate the flexibility of this new mechanism via examples.

1 Introduction

Coordination languages and models aim at providing mechanisms and languages for developing distributed applications in which the description of the internal behaviour of the active components and the description of their interdependencies are distinct and separated. In order to support this separation, general interaction models have been developed; one of the main approaches consists of separating the computation from the interaction information, and locating the former inside the active components and the latter inside a so-called coordination infrastructure. This approach is called data-driven coordination when the interaction information is stored inside a shared data space. Linda [Gel85] and its dialects [Sun02,WMF98,NFP98,MPR01] are the most prominent representatives of this family of coordination languages.

The Linda coordination model is based on a common repository where the interacting components introduce and retrieve the coordination information: the data inside the repository are tuples (ordered sequences of data) and they

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

are retrieved using a pattern matching mechanism. Processes willing to collect tuples indicate with a template the structure of the tuples they are interested in: the template indicates the number of fields and the exact content for some of these fields. A tuple inside the repository satisfies the template if it has the requested number of fields and it contains, at least in the indicated fields, the requested data.

The native coordination model does not include any support for controlling the access to the data. For example, any process able to access the repository is also able to introduce, retrieve and consume any of the tuples inside the repository itself. Recent distributed applications such as Web Services, applications for Mobile Ad Hoc Networks (MANETs), Peer to Peer Applications (P2P) are inherently open to processes, agents, components that are not known at design time. When the Linda coordination model is exploited to program the coordination inside this class of applications (see e.g. WSSecSpaces [LZ04] for Web Services, Lime [MPR01] in the context of MANETs and PeerSpaces [Zav03] for P2P applications) it is convenient to extend the native coordination model with mechanisms for controlling the access, for authenticating the producer/consumer of data, for ensuring the secrecy of information, and so on.

Some extensions have been already presented in the literature. The Klaim model [NFP98] (Kernel Language for Agent Interaction and Mobility) exploits classic access control policies to manage the access of processes to tuple spaces. Access permissions describe the operations that each process may perform on each of the available tuple spaces. In open systems, especially in those with a high level of dynamicity, the managing of these information may be a critical task, mainly because the system should support a rapid and sometimes uncontrolled evolution of the agent community. More precisely, new agents may frequently enter the system, as well as old agents may rapidly exit, in an uncontrolled manner. Moreover, in some applications it could be useful to have a finer grained control, e.g., at the level of tuples and not at the level of spaces. For example, we may want to ensure that an agent cannot read tuples with a private content, but it can read all of the other tuples.

SecOS [VBO03] follows a quite different approach. The access rights are not associated to the agents, but all control information are stored inside the data. More precisely, SecOS supports two forms of locks which are called *symmetric* and *asymmetric*. The former exploits the same key to protect and access the information, while the latter uses a pair of keys, one to protect and another one to access. This two locking techniques can be applied to protect either one single field inside a tuple or the whole tuple. In the first case the used locks are called *Field-locks*, while in the second one, they are called *Object-locks*.

A more recent proposal, called `SecSpaces` [BGLZ03], continues the approach initiated by SecOS by refining its access control policies. More precisely, `SecSpaces` refines the idea of object locks while field locks are not modeled. However, this is not a limitative approach because field locks can be easily encoded in our model. For example, in SecOS it is not possible to discriminate between the non-destructive readers and the destructive consumers of a tuple, and there is neither the possibility to avoid a reader of a tuple to re-produce exactly that tuple. In `SecSpaces`, on the other hand, it is possible to associate to the tuples two different access control information, one to be used for non-destructive read and the other one for destructive input operations. Moreover, when a tuple is accessed in `SecSpaces`, the reading process receives only the coordination information inside the tuple and not the access control information; thus, it is unable to reproduce exactly that tuple.

`SecSpaces` essentially permits to associate to the tuples two pairs of information, that now we call entries. Each pair (p, k) contains a symmetric access information p (called symmetric partition) and an asymmetric access information k (called asymmetric partition). For both kind of access information a default value is defined, thus processes can also explicitly set only one or none of them. In order for a reader to access an information, it must demonstrate the knowledge of p as well as of $\overline{k}$, the access information corresponding the asymmetric partition k. Two of these pairs (p, k) are used; the first one is considered in the case the reader is willing to perform a non-destructive read operation, and the second is considered for destructive input operations.

In this paper we investigate, in the context of `SecSpaces`, the problem of modifying during the lifetime of the application the access control policies. This is particularly useful in applications where the participants may become able to access new resources because, e.g., they pay for it, or in applications where processes may access the shared resources according to an associated level of trust, and this level may increase or decrease according to their run-time behaviour.

Tha native `SecSpaces` model is not particularly suited to support applications with this dynamic aspects. For example, if two separated groups of users decide at run time to join in a unique group, it is necessary to explicitly modify the access control information stored inside the entries currently used from both those groups of users. An alternative solution is to permit the users to exploit more expressive access rights indicating the intention to atomically access the data of both groups. Suppose, on the other hand, that it is necessary to exclude at run time a user from a group because its level of trust is decreased. In this context, to exclude a user means that it should have no more access to the new entries exchanged among the users of the group. Also in this case, the native `SecSpaces` model requires to explicitly modify the access control information stored inside the entries of the group.

The solution that we present consists of extending `SecSpaces` with the possibility to: i) create, at run time, fresh symmetric partitions, ii) combine partitions. We can combine partitions by using the merge operator, denoted by ":", whose meanings depends on the operation the process is performing:

- when a process perform a data-retrieval operation by using the merge of partitions p and p', that is $p : p'$, it implicitly access the partitions p and p';

- when a process inserts an entry into the space by combining partitions p and p', that is $p : p'$, that entry will be visible by processes that have access to one among the p and p' partitions.

In the paper we demonstrate that this simple extension of the `SecSpaces` coordination model is expressive enough to add a significant level of dynamicity. For example, in order to exclude users from a group that exchange entries using the partition p, it is sufficient to produce a new partition p', to distribute the knowledge of p' only to the users that remain inside the group, and to start using: i) the new partition p' to exchange new entries, and ii) the combination $p : p'$ to access all the entries produced by the group (all the new and all those ones produced before the group restriction).

It is worth noting that these extensions involve only the symmetric partitions and not the asymmetric ones. This follows from the fact that the two kinds of partitions are intended to represent different kinds of information. The symmetric partitions are used to limit the access to entries, while the asymmetric partitions are used to authenticate the producer/consumer of an entry. To this aim, the asymmetric partitions are intended to be produced and distributed using off-line mechanisms (such as standard public key infrastructures PKI) and not the mechanisms provided by the `SecSpaces` coordination model.

The remainder of the paper is structured as follows: in Section 2 we recall the `SecSpaces` coordination model which is defined in terms of a process calculus, in Section 3 we extend the process calculus introducing the possibility to create new partitions as well as combining partitions, in Section 4 and 5 we describe some applications that demonstrate the flexibility of the new extensions in order to model several access control policies, and in Section 6 we report some conclusive remarks.

2 The `SecSpaces` coordination model

`SecSpaces` [BGLZ03] is a coordination model that supports secure data-driven coordination in open environments. `SecSpaces` extends Linda [Gel85] by introducing some forms of control of the accesses to the entries stored in the coordination space, that permit for example to authenticate/identify the producer of an entry or its reader/consumer.

The coordination primitives of `SecSpaces` are the classical ones of Linda: $out(e)$, $in(t)$ and $rd(t)$. The output operator $out(e)$ inserts an entry e in the space. Primitive $in(t)$ is the blocking input operator: when an occurrence of an entry e matching with the template t is found in the space, it is removed and its content is returned. The read primitive $rd(t)$ is similar to $in(t)$, but in this case the entry e is not removed from the space.

In order to express access permissions on entries `SecSpaces` extends the

Linda tuples by decorating them with special control fields, namely *partition* and *asymmetric partition* fields. The former ones logically partition the space, while the latter ones provide a mean to discriminate between the write and the read/remove access permissions on each entry.

Let $Mess$, ranged over by m, n, ..., be an infinite set of messages, $Partition$, ranged over by c, c_t, ..., be the set of partitions and $APartition$, ranged over by k, k', k_t, ..., be the set of asymmetric partitions. We also assume that $Partition$ (resp. $APartition$) contains a special default value, say # (resp. ?), used to allow any agent to access the space. Let "$\overline{\cdot}$: $APartition \rightarrow APartition$" be a function, defining the co-key relationship, such that $\overline{?} = ?$ and if $\overline{k} = k'$ then $\overline{k'} = k$.

Access permissions on entries are expressed by the control fields; in order to discriminate between the rd and the in access permission, entries have two occurrences of control fields, one associated to in operations and the other one to the rd operations. Differently from entries, templates have only one occurrence of control fields that is not associated to a specific operation: they are dynamically associated to the operation the agent is willing to perform (i.e., rd or in).

The set $Entry$ of entries, ranged over by e, e', ..., is defined as follows:

$$e = \langle \vec{d} \rangle^{[c]_{rd}[c']_{in}}_{[k]_{rd}[k']_{in}}$$

where $c, c' \in Partition$, $k, k' \in APartition$ and the tuple of data $\vec{d}$ is a term of the following grammar:
$\vec{d} ::= d \mid d; \vec{d}$,
$d ::= m \mid c \mid k$.
A *data field* d can be a message, a partition or an asymmetric partition.

We define $\tilde{\cdot}$ as the function that, given an entry e, returns its tuple of data, i.e., if $e = \langle \vec{d} \rangle^{[c]_{rd}[c']_{in}}_{[r]_{rd}[r']_{in}}$, $\tilde{e} = \vec{d}$.

The set $Template$ of templates, ranged over by t, t', ..., is defined as follows:

$$t = \langle \vec{dt} \rangle^{[c_t]}_{[k_t]}$$

where $c_t \in Partition$, $k_t \in APartition$ and $\vec{dt}$ is a term of the grammar
$\vec{dt} ::= dt \mid dt; \vec{dt}$,
$dt ::= d \mid null$.
With respect to entries, data fields used by templates can also be set to an additional value ($null$) that denotes the wildcard: the wildcard is used to match with all field values.

Definition 2.1 Matching rule and return value – Let $e = \langle d_1; d_2; \ldots; d_n \rangle^{[c]_{rd}[c']_{in}}_{[k]_{rd}[k']_{in}}$ be an entry, $t = \langle dt_1; dt_2; \ldots; dt_m \rangle^{[c_t]}_{[k_t]}$ be a template and $op \in \{rd, in\}$ be an operation. Let c_e and k_e be the control fields of e associated to op, we say that e *matches*$_{op}$ t if the following conditions hold:

(i) $m = n$

(ii) $dt_i = d_i$ or $dt_i = null$, $1 \leq i \leq n$

(iii) $c_e = c_t$

(iv) $\overline{k_e} = k_t$.

If a *rd* or *in* operation with template t is performed on a matching entry e, only the data fields (and no control field) are returned, i.e., the return value is $\tilde{e}$.

Conditions (i) and (ii) rephrase the classical Linda matching rule, that is test if e and t have the same arity and if each data field of e is equal to the corresponding field of t or if this latter one is set to wildcard. Condition (iii) tests that the partition field of the entry –associated to the operation *op*– is equal to that of the template. Condition (iv) checks that the asymmetric partition field of the template corresponds to the co-key of the asymmetric partition field of the entry associated to the operation *op*. Finally, a comment about return value of the data-retrieval operations: it does not include the control fields of the matching entries but only the data stored inside the tuple of data fields. In this way, new access permissions can be acquired only by performing read/input operations of entries containing partition or asymmetric partition values inside the tuple of data.

Partition fields can be viewed as a special kind of data fields that do not accept wildcard in the matching evaluation. Partition fields logically partitionate the space (each partition contains the entries having a specific partition value as partition field that identifies the partition) and the access to a partition is restricted to only those processes that know the partition identifier. Indeed, in order to perform an operation on a partition, processes must know the partition field identifying that specific partition. However, in order to allow any process to interact with each other via `SecSpaces` primitives, a special default value # of the partition field, that every one can use, has been defined. Similarly, a default value known by any process has also been defined for asymmetric partition fields (denoted by "?").

The asymmetric partition fields, differently from partitions, make it possible to discriminate between the write and the read/remove permission of an entry, simply by exploiting the different needed knowledge to produce or to read/remove an entry. For instance, in order to read an entry having asymmetric partition field set to k the process must use $\overline{k}$ as asymmetric partition field of the template, that can be an unknown value for the producer of that entry (because in only k is needed). Therefore, following the same idea of partitions, properly distributing these values we can assign processes the permission to perform a subset of possible operations on that entry.

3 `SecSpaces` with combined partitions

In this section we propose an extension of `SecSpaces` supporting dynamic composition of partitions. More precisely, we introduce a new merge operator on partitions (denoted by ":") that can be used to express in which partitions an entry should be accessible (when writing) or in which partitions to perform the search of a matching entry (when reading or consuming). The language we are going to presents extends [BGL03] by introducing the merge operator on partitions and an operator for the generation of new names for partitions. In Section 3.1 we present the extended language and the new definition of the matching rule, while Section 3.2 introduces the corresponding semantics.

3.1 The language

In this section the extended language is presented. More precisely, we first describe how partitions are extended with the new merge operator and then we formalize system configurations, that are composed of: processes exploiting `SecSpaces` coordination primitives and the state of the shared tuple space.

Let $CPartition$, ranged over by p, p', …, be the set of possible combinations of partitions defined by the following grammar:
$p ::= c \mid p : p$.
The combination of partitions can be a partition value or the merge of partitions: "$c : c'$" represents the merge of the partitions identified by c and c'. We also define $ps : CPartition \longrightarrow \mathcal{P}(Partition)$ as the function that, given a combination of partitions, returns the set of partitions it contains, whose definition is the following: [1]

$$ps(c) = \{c\}, \quad ps(p_1 : p_2) = ps(p_1) \cup ps(p_2).$$

Entry and template structure change in the symmetric partition fields that now contains, instead of a partition $c \in Partition$, a combination of partitions $p \in CPartition$. An entry $e = \langle \vec{d} \rangle^{[p]_{rd}[p']_{in}}_{[k]_{rd}[k']_{in}}$ means that it is available (i.e. it appears) in the partitions contained in $ps(p)$ and in $ps(p')$ for the rd and the in primitives, respectively. On the other hand, a template $t = \langle \vec{d} \rangle^{[p_t]}_{[k_t]}$ means that the matching entry must be available in at least one partition in $ps(p_t)$. The definition of the matching rule between entries and templates follows.

Definition 3.1 Matching rule with combined partitions – Let $e = \langle d_1; d_2; \ldots; d_n \rangle^{[p]_{rd}[p']_{in}}_{[k]_{rd}[k']_{in}}$ be an entry, $t = \langle dt_1; dt_2; \ldots; dt_m \rangle^{[p_t]}_{[k_t]}$ be a template and $op \in \{rd, in\}$ be an operation. Let p_e be the combined partition of e associated to op, we say that e $matches_{op}$ t if conditions (i), (ii) and (iv) of Definition 2.1 and the following condition hold:

(iii)' $ps(p_e) \cap ps(p_t) \neq \phi$.

Condition (iii)' replaces (iii) of Definition 2.1 substantially checks that there is at least one partition in which e is available that is contained in the set of partitions indicated by the template.

[1] We assume that associative property for the merge operator holds, that is $(c : c') : c'' = c : (c' : c'')$.

When a matching entry is accessed from a removal operation, it is removed by any partition in which it appears. The idea is that a process can exploit the merge mechanism to produce an entry that must appear in more than one partition by performing just one *out* operation; if a process needs to produce more than one occurrence of the entry, it must perform an *out* operation for each occurrence it needs.

It is worth noting that the same mechanisms can be implemented by introducing, in the `SecSpaces` model, the non-blocking *rdp* and *inp* primitives corresponding to the *rd* and *in*, and a transaction mechanism. Non-blocking data-retrieval primitives have the same behaviour of the corresponding blocking ones in the case a matching entry is available, while in the opposite case they immediately return with a failure value indicating the absence of a matching entry. When processes need to perform a data-retrieval primitive accessing more than one partition, say c and c' (the extension to a generic number is straightforward), they can exploit non-blocking primitives by alternatively performing the access to partition c and to c' until a matching entry is found. On the other hand, in order to produce an entry that should appear in more than one partition, an occurrence of the entry can be introduced in each partition in which it should be available. Transactions are necessary because, in order to consume that entry, the *in* operations should perform atomically the removal of the matching entry from each partition in which it appears.

The entries stored in the TS are represented as members of parallel composition as processes. Let Var, ranged over by x, y, ..., be the set of data variables. In the following, we use $\vec{x}$, $\vec{y}$, ..., to denote finite sequences $x_1; x_2; \ldots; x_n$ of data variables.

System configurations, ranged over by A, B, ..., and processes, ranged over by P, Q, ..., are defined as follows:

$A, B, \ldots ::=$	systems	
e		entries
P		processes
$A \mid A$		parallel composition
$(\nu\, c)\, A$		restriction

$P, Q, \ldots ::=$	processes	
$\mathbf{0}$		null process
$out\ e.P$		output
$rd\ t(\vec{x}).P$		read
$in\ t(\vec{x}).P$		input
$P \mid P$		parallel composition
$!P$		replication
$(\nu\, c)\, P$		restriction

A system can be an entry, a process or the parallel composition of entries and processes. $(\nu\, c)\, A$ means that the partition name c is bound in the system A. A process can be a terminated program $\mathbf{0}$, a prefix form $\mu.P$, the parallel composition of two programs, the replication of a program or it can contain a partition name whose scope is restricted to the process. The prefix μ can be one of the following classical Linda operations: i) *out* e, that writes the entry e in the TS; ii) $rd\ t(\vec{x})$, that given a template t reads a matching entry e in the TS and stores the return value in $\vec{x}$; iii) $in\ t(\vec{x})$, that given a template t consumes a matching entry e in the TS and stores the return value in $\vec{x}$. A process $P \mid Q$ is the parallel composition of two processes P and Q behaves as two processes running in parallel. Recursive process are expressed by using the replication operator $!P$, whose meaning is the parallel composition of infinite copies of P. Finally, processes can have bound names (partitions): $(\nu\, c)\, P$ means that the partition name c is bound in the process P, hence it is known only by P.

In the following, we use $P[d/x]$ to denote the process that behaves as P in which all occurrences of x are replaced with d. We also use $P[\vec{d}/\vec{x}]$ to denote the process obtained by replacing in P all occurrences of variables in $\vec{x}$ with the corresponding value in $\vec{d}$, that is $P[d_1; d_2; \ldots; d_n/x_1; x_2; \ldots; x_n] = P[d_1/x_1][d_2/x_2] \ldots [d_n/x_n]$.

We say that a system is *well formed* if each $rd/in\ \langle\vec{dt}\rangle^{[c]_{rd}[c']_{in}}_{[k]_{rd}[k']_{in}}(\vec{x})$ operation is such that the variables $\vec{x}$ and the tuple of data $\vec{dt}$ have the same arity. Let $fn(A)$ and $fv(A)$ be the functions that given a system A return the set of names that syntactically occur in A and the set of free variables in A, respectively. We say that a system is *closed* if it has no free variable. In the following, we consider only systems that are closed and well formed; we denote with *System* the set of such systems.

3.2 Semantics

In this section we present the semantics of systems by defining a *structural congruence* over systems and by mapping them on a reduction relation that describes how the system is reduced after one step of computation.

Table 1 describes a relation (structural congruence) indicating syntactic differences between systems that do not influence the behaviour of processes. More precisely, identity (i), reflexive (ii) and transitive (iii) relations hold, the order of the systems parallel composition is not relevant (iv), associative relation holds (v), portions of system can be replaced with other ones structurally equivalent (vi), null processes have no influence on the behaviour of the system (vii), replication operator $!P$ corresponds to an infinite parallel composition of P (viii), (ix), (x) and (xi) represent the scope laws and, finally, (xii) describes the alpha conversion (i.e. choice of bound names is irrelevant). The structural congruence over systems is defined as the smallest congruence satisfying rules (i), …, (xii).

Table 2 contains the system reduction rules. Rules (1), (2) and (3) describe

(i)	$A \equiv A$	(ii)	$\frac{B\equiv A}{A\equiv B}$
(iii)	$\frac{A\equiv B \quad B\equiv C}{A\equiv C}$	(iv)	$A \mid B \equiv B \mid A$
(v)	$(A \mid B) \mid C \equiv A \mid (B \mid C)$	(vi)	$\frac{A\equiv A'}{A\mid B\equiv A'\mid B}$
(vii)	$A \mid \mathbf{0} \equiv A$	(viii)	$!P \equiv P\ \mid !P$
(ix)	$(\nu\, c)\,(\nu\, c')\, A \equiv (\nu\, c')\,(\nu\, c)\, A$	(x)	$(\nu\, c)\,\mathbf{0} \equiv \mathbf{0}$
(xi)	$(\nu\, c)\,(A \mid B) \equiv (\nu\, c)\, A \mid B$, if $c \not\in fn(B)$		
(xii)	if A can be turn to B by alpha-conversion then $A \equiv B$		

Table 1
Structural equivalence over systems

the three prefix operators *in*, *rd*, and *out*, respectively. More precisely: (1) shows that the process *in* $t(\vec{x}).P$ can perform the input if there exists an entry e currently available in the TS that matches the template t and, in this case, e is removed from the TS and the process behaves as $P[\tilde{e}/\vec{x}]$; (2) shows that the process *rd* $t(\vec{x}).P$ can perform the read operation if there exists an entry e currently available in the TS that matches the template t and, in this case, the process behaves as $P[\tilde{e}/\vec{x}]$, and (3) shows that *out* $e.P$ produces in one step a new occurrence of the entry e into the TS and then the process behaves as P. Rules (4), (5) and (6) describe the behaviour of processes running in parallel, that we can replace at any time a system with another one structurally congruent, and how the interaction is restricted to the processes within the scope operator, respectively. It is worth noting that rules (xi) and (xii) of Table 1 and rules (1) and (2) allow us to implement the scope extrusion of partition names as originally proposed in the π-calculus [**?**].

$$
\begin{array}{llll}
(1) & \dfrac{e\ matches_{in}\ t}{e \mid in\ t(\vec{x}).P \longrightarrow P[\tilde{e}/\vec{x}]} & (2) & \dfrac{e\ matches_{rd}\ t}{e \mid rd\ t(\vec{x}).P \longrightarrow e \mid P[\tilde{e}/\vec{x}]} \\[2ex]
(3) & out\ e.P \longrightarrow e \mid P & (4) & \dfrac{A \longrightarrow A'}{A \mid B \longrightarrow A' \mid B} \\[2ex]
(5) & \dfrac{A \equiv B \quad B \longrightarrow B'}{A \longrightarrow B'} & (6) & \dfrac{A \longrightarrow A'}{(\nu\, c)\, A \longrightarrow (\nu\, c)\, A'}
\end{array}
$$

Table 2
Transition system

4 Group communication examples

The aim of this section is to describe scenarios where the extension of `SecSpaces` we propose can be exploited to manage important task. In previous papers [BGL03,BGLZ03] we have already proved that some security properties in the interaction among processes can be guaranteeted. In particular, we have shown that it is possible to guarantee data secrecy, producer and receiver authentication of an entry and data availability. Here we now recall recall the idea, that we have already introduced in those papers, to implement a secure group communication, where only entities of the group can access the entries used in the group communication. In that work, we have considered a simple solution in which the entities in the group share a secret partition, say c (that can be considered as a session partition) that is used to restrict the access to the exchanged entries: any communication is done by using entries and templates having c as partition field. In this way, no other process can read, remove or produce an entry used to realize the group communication because c is not a public value, thus guaranteeing the privacy of the exchanged data. An example of secure group communication follows (for the sake of simplicity asymmetric partition fields, assumed that they are set to default value, are omitted): $(\nu\, c)\, rd\ \langle null \rangle^{[c]}(x).out(\langle x \rangle^{[c]_{rd}[c]_{in}}) \mid out(\langle d \rangle^{[c]_{rd}[c]_{in}})$. A proposal of how to distribute the partition value c in a secure way (assuming that asymmetric partition fields are properly distributed as explained in the Introduction), to each process of the group, is described in Section 4.1.

While it is easy to manage the insertion of new entities in the group, because it can be done simply by transmitting (in a secure way) the secret partition of the group to the entity that is willing to enter (technically this is obtained by scope extrusion, see Section 3.2), it is more complicated to manage the removal of certain entities from the group communication, that is an usual function of group key management systems, because a new partition should be created and distributed to the processes in the group except the one to be removed. A more general problem is to restrict the access to certain entries to a subgroup of entities in the group. Section 4.1 explains how to manage the group restriction by exploiting the merge operator. Finally, Section 4.2 describes how to combine partitions in order to manage, at run-time, the coordination among two (or more) independent groups of processes. More precisely, we intend to provide a support for those applications that need to publish some data and make them available to more than one group of users, or to control the flow of exchanged data among different groups.

4.1 Group restriction

Let $G = \{P_1, \ldots, P_n\}$ be a group of processes that exploit a shared (and private to the group) partition c to communicate with each other in the group, and $G_p = \{P_i, P_{i+1} \ldots, P_s\} \subseteq G$ be a subset of the processes in G.

The problem we consider in this section is to:

i) provide a secure group communication for groups G and G_p, and

ii) allow processes in G_p (that should be considered privileged processes of G) to perform data-retrieval operations that can atomically access entries available either at the whole group or to privileged processes.

The solution we propose is to define, besides c, a new partition c' that is a shared and private value of privileged processes that exploit it in order to restrict the communication at the level of group G_p. More formally, the system configuration we want to define is the following:

$$(\nu\, c)\, (P_1 \mid \ldots \mid P_{i-1} \mid (\nu\, c')\, (P_i \mid \ldots \mid P_s) \mid \ldots \mid P_n).$$

Statements i) and ii) hold because: 1) each process in G can communicate with each other in the group by using c as partition field of the entries/templates and, in addition, privileged processes can also restrict the communication to the group G_p by inserting entries with a partition set to c', e.g., $out(\langle d\rangle^{[c']_{rd}[c']_{in}}$; 2) privileged processes in one step can access the two partitions, simply by exploiting the merge operator, indicating $c : c'$ in the partition field of the template, for instance $rd\ \langle null\rangle^{[c:c']}(x)$. A particular instance of this problem is the removal of some users from the group: in this case to combine the partition c with the new partition c' allow us to atomically access the new entries as well as to those ones produced before the removal of such users.

The following example describes a possible way to implement the secure distribution of c' to the processes in G_p.

Example 4.1 Given a process P_i that generates the new partition c', we consider a set of $s - i$ asymmetric partitions k_{ij} with j such that $j : i+1 \leq j \leq s$. We assume that for every $i + 1 \leq j \leq s$, k_{ij} is known only by the process P_i while $\overline{k_{ij}}$ is known only by P_j. The group restriction can be implemented by exploiting the following distribution protocol:

$$(\nu\, c)\, (P_1 \mid \ldots \mid P_{i-1} \mid P'_i \mid \ldots \mid P'_s \mid \ldots \mid P_n)$$

where

$$P'_i = (\nu\, c')\, \mathit{out}(\langle c' \rangle^{[c]_{rd}[c]_{in}}_{[k_{i(i+1)}]_{rd}[k_{i(i+1)}]_{in}}).\mathit{out}(\langle c' \rangle^{[c]_{rd}[c]_{in}}_{[k_{i(i+2)}]_{rd}[k_{i(i+2)}]_{in}}) \ldots$$

$$.\mathit{out}(\langle c' \rangle^{[c]_{rd}[c]_{in}}_{[k_{is}]_{rd}[k_{is}]_{in}}).P_i, \text{ and}$$

$$P'_j = \mathit{in}\ \langle \mathit{null} \rangle^{[c]}_{\overline{[k_{ij}]}}(x).P_j, \text{ for } i+1 \leq j \leq s.$$

P'_i creates a new partition c' and then distributes it to the processes in G_p thus reaching the configuration $(\nu\, c)\, (P_1 \mid \ldots \mid P_{i-1} \mid (\nu\, c')\, (P_i \mid \ldots \mid P_s) \mid \ldots \mid P_n)$. It should be clear that the entry $\langle c' \rangle^{[c]_{rd}[c]_{in}}_{[k_{ij}]_{rd}[k_{ij}]_{in}}$ can be produced only by P'_i and that only P'_j can access that entry, for any $i+1 \leq j \leq s$.

4.2 Group coordination

In this section we discuss how to exploit the merge operator in order to support the coordination among different groups, such as to distribute data to more than one group.The idea we follow is that a process (a new one or one elected among those in the groups) takes the role of *coordinator*; it knows the partition associated to each group and, by merging all (or part of) group partitions, can insert entries that will be accessible to all (or part of) groups as well as read and remove all entries generated by all groups.

We consider the case where two groups have to be coordinated, the extension to an arbitrary number of groups is straightforward. Let $G = \{P_1, \ldots, P_n\}$ and $G' = \{P'_1, \ldots, P'_k\}$ be two groups of processes that exploit a shared (and private to the group) partition c and c' to communicate with each other of the group, respectively.

We describe in the details the case in which the coordinator provides to groups a way to insert data that should be accessible in both groups (i.e. partitions c and c'). In order to avoid to give direct access to the partitions of the other groups, we assume that each group has an identifier and that knows the group identifiers of the other groups; we use g and g' for the group G and G', respectively. When a process of the group (e.g., G) is willing to produce an entry that should be available in other groups, it produces an entry in its space that contains, in the tuple of data fields, the identifiers of the group in which the entry should be available and the data d it is willing to publish (e.g., $\mathit{out}(\langle g'g, d \rangle^{[c]_{rd}[c]_{rd}})$). The coordinator removes such entries, logs the request and then publishes the entry. The coordinator process definition follows:

$$!(\mathit{in}\ \langle \mathit{null}; \mathit{null} \rangle^{[c:c']}(x_1; x_2).\mathit{LogReq}(x_1, x_2).\mathit{out}(\langle x_2 \rangle^{[P(x_1)]_{rd}[P(x_1)]_{in}})),$$

where *LogReq* registers the request and $P(\mathit{grp})$ returns the expression that merges the partitions identified in grp (e.g., $P(g'g) = c : c'$). The role of coordinator can also be extended by also managing access rights, i.e. checking if a group can write in a specified group. It is worth noting that the same approach can be used if, instead of c and c', we have to manage two combinations of partitions, in this case we distribute information that are already available in more than one partition.

5 A system with multi-level security

The flexibility and the expressiveness of the proposed extension is proved in this section by implementing a multilevel system where a process having access to a certain level l can also access to each level higher than l.

In order to be as general as possible, we consider that an hierarchical tree describes the security levels of the system, represented by nodes, and their (partial) ordering relation, that is: each node (level) is higher than each child.

The model we intend to encode in `SecSpaces` allows, given a level l, to: i) produce a new datum at the specified level l, and ii) perform a data-retrieval operation that can access datum available either at the level l or at the levels higher than l. In other words, in this system, a datum produced by processes accessing level l are available in l and in all levels l' for which l is higher than l'. In this way, each leaf of the hierarchical tree can access each level in the path leaf-root and may be considered as the level with maximum privileges among those in the path.

In order to encode the system we proceed in two steps: i) we introduce a new operator for sub-partition the partitions that will be exploited to encode the system, and ii) we encode the introduced operator by exploiting the proposed merge one.

In order to introduce sub-partition operator we extend the syntax of partitions:
$p ::= c \mid p : p \mid p \rightarrow p.$
Given a partition p_1, the sub-partition operator allow us to introduce new partitions ($p_1 \rightarrow p$, for any p) which are by definition sub-partitions of p_1. In general, a partition is a sub-partition of another one if the name of the latter is a prefix of the former one (e.g., $p_1 \rightarrow p_2 \rightarrow p_3$ is a sub-partition of $p_1 \rightarrow p2$ as well as of p_1).

The idea we follow is that partitions represent the nodes (i.e. levels) of the hierarchical tree and that the sub-partition operator allows us to express the relation between nodes and ancestors: each partition, say p_1, is an ancestor of each sub-partition of p_1, for instance $p_1 \rightarrow p$ expresses that p_1 is an ancestor of $p_1 \rightarrow p$ (e.g., $p \rightarrow c$ has father p) and then p_1 has a level higher than $p_1 \rightarrow p$. The definition of the function ps is extended with the following case:

$$ps(p_1 \rightarrow c) = \{p_1 \rightarrow c\} \cup ps(p_1).$$

By using the merge and sub-partition operators to combine partitions, the matching rule (see Definition 3.1) allows to match entries that appear either in the partition specified by the template or in each partition with higher level.

In order to encode the system, we assume that the knowledge of the security levels identifiers (i.e. partitions which also represent hierarchical level relationship via the → operator) has been properly distributed to the processes; this phase can be done by exploiting asymmetric partition fields (for more details see [BGLZ03]). The multilevel system provides the following primitives, whose encoding is:

- $out(\vec{d}, p) = out(\langle\vec{d}\rangle^{[p]_{rd}[p]_{in}})$, that inserts the datum $\vec{d}$ in the level p,
- $rd(\vec{dt}, p) = rd\langle\vec{dt}\rangle^{[p]}(\vec{x})$, that reads a datum matching with $\vec{dt}$ by using access level p, and
- $in(\vec{dt}, p) = in\langle\vec{dt}\rangle^{[p]}(\vec{x})$, that consumes a datum matching with $\vec{dt}$ by using access level p.

It should be clear that the system so defined encodes the multi-level system described above. For instance, a process having access to the level $p_1 \rightarrow p$ can read/consume any data available at a level in $ps(p_1 \rightarrow p)$.

Finally, here we show how to encode the sub-partition operator by exploiting only the merge operator. The only change regards the partition field value used by the `SecSpaces` primitives, that is: i) in the *out* primitive we keep the same partition p; in the case it contains the sub-partition operator, it will be considered as part of the partition name, and ii) in the rd/in primitive, p is encoded as the application of the merge operator to each partition determined with the extended definition of $ps(p)$ considered in this section. For instance, $out(\vec{d}, (c_1 : c_2) \rightarrow c_3)$ is encoded with $out(\langle\vec{d}\rangle^{[(c_1:c_2)\rightarrow c_3]_{rd}[(c_1:c_2)\rightarrow c_3]_{in}})$ that produces an entry in a partition that is to an higher level than the one used by $rd(\vec{dt}, ((c_1 : c_2) \rightarrow c_3) \rightarrow c_4)$, encoded with $rd\langle\vec{dt}\rangle^{[(((c_1:c_2)\rightarrow c_3)\rightarrow c4):((c_1:c_2)\rightarrow c_3):(c_1:c_2)]}(\vec{x})$. It is worth noting that the knowledge necessary for the encoding can be derived (by using the extended definition of ps) from the datum stored in the primitives, therefore we do not need to modify the knowledge of the processes.

6 Conclusion and future work

In this paper we have introduced an extension of `SecSpaces` in order to provide a solution for problems such as group communication management or interactions among processes with different levels of access permission. The flexibility of the proposed solution has been proved by describing several examples in which it is necessary a support for run-time managing of the scope of entries.

Finally, as future work, we intend to compare the extended version of `SecSpaces` with some multilevel systems, such as [KMM] that is a multilevel system supporting one-way flow that has some similarities with the model described in Section 5.

References

[BGL03] Mario Bravetti, Roberto Gorrieri, and Roberto Lucchi. A formal approach for checking security properties in secspaces. In Riccardo Focardi and Gianluigi Zavattaro, editors, *Electronic Notes in Theoretical Computer Science*, volume 85. Elsevier, 2003.

[BGLZ03] Nadia Busi, Roberto Gorrieri, Roberto Lucchi, and Gianluigi Zavattaro. Secspaces: a data-driven coordination model for environments open to untrusted agents. In Antonio Brogi and Jean-Marie Jacquet, editors, *Electronic Notes in Theoretical Computer Science*, volume 68. Elsevier, 2003.

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[KMM] Myong H. Kang, Andrew P. Moore, and Ira S. Moskowitz. Design and Assurance Strategy for the NRL Pump. In *Proc. IEEE High-Assuruance Systems Engineering Workshop (HASE97).*

[LZ04] Roberto Lucchi and Gianluigi Zavattaro. WSSecSpaces: a Secure Data-Driven Coordination Service for Web Services Applications. In *Proc. of ACM Symposium on Applied Computing (SAC'04).* ACM Press, 2004.

[MPR01] A. Murphy, G. Picco, and G.-C. Roman. A middleware for physical and logical mobility. In *21st International Conference on Distributed Computing Systems*, pages 524–533, 2001.

[NFP98] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, May 1998. Special Issue: Mobility and Network Aware Computing.

[Sun02] Sun Microsystems, Inc. *JavaSpaces^{TM} Service Specification*, 2002. http://www.sun.com/jini/specs/.

[VBO03] Jan Vitek, Ciarán Bryce, and Manuel Oriol. Coordinating Processes with Secure Spaces. *Science of Computer Programming*, 46:163–193, 2003.

[WMF98] P. Wyckoff, S.W. McLaughry, and D.A. Ford. TSpaces. *IBM System Journal*, August 1998.

[Zav03] Gianluigi Zavattaro. PeerSpaces: Data-driven Coordination in Peer-to-Peer Networks. In *Proc. of ACM Symposium on Applied Computing (SAC'03)*, pages 380–386. ACM Press, 2003.