# A formal approach for checking security properties in SecSpaces ⋆

Roberto Lucchi, Mario Bravetti and Roberto Gorrieri

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

**Abstract**

SecSpaces is a Linda-like coordination model whose aim is to provide a support for secure coordination in Open System applications. Substantially it provides a methodology to restrict the access to the objects stored in the shared dataspace. In this paper we introduce a formal language for representing systems interacting via SecSpaces primitives and its operational semantics. Moreover in this context we consider a notion of observational equivalence, namely testing equivalence. In order to evaluate the adequacy of the model for limiting the access to the shared dataspace, we present some examples of interaction protocols that can be used to obtain some security properties (e.g., authentication or privacy of a datum).

## 1 Introduction

Emerging technologies are moving to support *Open Systems* applications. Such applications are characterized by the fact that the entities that will be involved in the applications as well as the system configuration are not known at the design time. Examples of open system applications are present in several contexts: i) wide area applications (Internet); ii) ad-hoc networks for mobile devices; iii) peer-to-peer systems.

The main contributions that gave rise to coordination as a new independent research field are based on the fact that we can view applications as composed of two orthogonal parts: the internal computation and the interaction. In particular, this concept was depicted by Carriero and Gelernter [CG92]. Due to the clear separation between the internal behaviour of the processes and their interaction, coordination languages represent a promising approach to support open systems applications. Indeed, several projects providing a

⋆ Work partially supported by MEFISTO Progetto "Metodi Formali per la Sicurezza e il Tempo"and Microsoft Research Europe.

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

support for Open Systems and self reconfigurable systems use a Linda-like [Gel85] data-driven coordination model. For instance, JavaSpaces [Sun02] and TSpace [WMF98] are two recent middlewares providing coordination for Java [GJS96] distributed programming proposed by Sun Microsystems and IBM, respectively. Both proposals exploit the so-called *generative communication* [Gel85]: a sender communicates with a receiver through a shared tuple space (TS for short), where emitted tuples are collected; the receiver can consume the tuples from the TS. Tuples in the TS have an existence which is independent of its producer, i.e. the agent that have generated a tuple in the TS does not have any form of control on that tuple; tuples in the TS become equally accessible to all agents that can remove or read any available tuple.

In open systems, applications cannot assume that all the involved entities are trusted, hence new aspects come into play. More precisely, each agent having the access to the TS can read/remove/manipulate each tuple stored in the TS. This means that there is no way to restrict the tuple access to a subset of all entities that can access the TS, e.g., the set of entities involved in a specific application. More in general, processes have no way to keep secret any tuple stored into TS.

The different solutions that have been proposed in order to solve this problem can be distinguished into two classes depending on their (orthogonal) goals: either they are based on mechanisms capable to ensure privacy of exchanged data (e.g., cryptography, [BN02]), or access control mechanisms [NFP98,VBO03,Pin92,MMU01,BGLZ02]. Here we will focus on the `SecSpaces` approach of [BGLZ02]. With respect to a technique following a classic control access mechanism (e.g., that used in Klaim [NFP98,NFP97]) where permissions describe which operations the entities can perform on the available tuple spaces, the `SecSpaces` proposal is based on the idea introduced by SecOS [VBO03]. SecOS does not associate access permissions to entities but rather follows the idea of a data-driven access control mechanism. More precisely, SecOS is a capability based system; access permissions are modeled by locks: there are both object locks referring to the whole object (a tuple) and field locks attached to a single field of the tuple. An agent can access an object (resp. field) only if it is able to provide the correct access key opening the attached lock. As shown in [BGLZ02], in SecOS, due to the way the read operation is encoded, there is no way to discriminate between the read and the remove permission; as illustrated in Section 5.5, this discrimination can be useful in some applications. In addition, each process able to read a datum is also able to reproduce that datum (read permission implies write permission). `SecSpaces` refines the access permissions of SecOS by making it possible to discriminate the permission of write, read and remove a specific tuple. Instead of using locks, `SecSpaces` decorates tuples with additional control fields of two types: partition fields and cryptographic fields. The former ones are just used to limit the access to a partition of the space: they do not provide a way to describe which operations an agent can perform on a specific partition.

The latter ones refine permission granularity at the level of tuples and make it possible to describe which operations an agent can perform on a tuple.

Comparing the data-driven approach of `SecSpaces` and SecOS to model access control with the classical one of Klaim, we observe that in the former the access permissions granularity is finer than in the latter: in `SecSpaces` and SecOS they are at the level of single tuples whilst in Klaim they are at the level of tuple spaces. It is worth noting that in Klaim we can obtain the same granularity of SecOS and `SecSpaces` by creating a new location (that contains a tuple space) for any entry. However, this approach should give rise to an uncontrollable proliferation of tuple spaces that should be available at the system startup, because Klaim security policies do not change dynamically (however, it has been recently presented MuKlaim [GP03] that implements the basic features of Klaim and that allows dynamic privileges acquisition). Finally, we note that, when using classic access control list mechanisms as done by Klaim, the permissions managing can be a difficult task if the open system is characterized by a high level of dynamicity. In [BGLZ02] the primitives of `SecSpaces` have been described by explaining their effect on the shared tuple space but a formal behavioral description of systems using such primitives has not been considered. The new contribution of this paper can be summarized in the following three points:

- we define a syntax for modeling open systems which make use of `SecSpaces` primitives and an operational semantics, i.e. how processes and space change during the computation. To be precise, the formal paradigm that we adopt is obtained by extending the Linda process algebra proposed in [BGZ98] with new ingredients needed for modeling the primitives of `SecSpaces`, i.e. entries and templates with a tuple structure and matching rules based on access fields;
- we define a notion of observational equivalence based on may testing [NH84] that allows us to compare the behavior of systems;
- we rephrase in our formal context some of the main security properties (e.g., secrecy, authentication) by exploiting observational testing equivalence. In particular, we rephrase the secrecy and message authentication properties by using the idea introduced in [AG99] and we show how other properties as producer and receiver authentication can be formalized by using a similar idea. We also provide some examples of property checking via the above introduced techniques on some simple interaction protocols.

The paper is structured as follows. Section 2. gives the syntax of system descriptions, Section 3. presents the operational semantics of `SecSpaces` and Section 4. defines testing equivalence that we exploit to formally prove basic safety properties. Section 5. presents the formal techniques to be used for checking the various kinds of security properties against system descriptions. Finally, Section 6. comments related works and concludes the paper.

## 2 Syntax

The section defines the syntax we use to represent systems, that is processes using `SecSpaces` coordination primitives and the shared space. We define *entries* to be the objects (tuples) that can be written in the TS and *templates* to be the data structures a process can use to find entries in the TS.

Differently from [BGLZ02], in this paper, for the sake of simplicity, we do not consider subtyping on entries but rather a simple matching rule on data fields similar to that used in Linda. Indeed, since the matching rule on data fields does not influence the access control mechanism which is exclusively based on control fields, this is not a limiting assumption. Moreover, in order to be as general as possible we model cryptographic fields in a more abstract way with respect to the original work on `SecSpaces` [BGLZ02]: we simply consider this field as composed by a single key, say $k$, that can match only with an associated co-key, say $\overline{k}$. We call this kind of control fields *asymmetric partition* fields because, as it will be clear in the following, the write permission to the asymmetric partition $k$ is allowed to each process that knows $k$, whilst the read permission to those that know $\overline{k}$. In this way, the cryptographic fields of [BGLZ02] represent a possible realization of asymmetric partition fields which exploits asymmetric cryptography.

Section 2.1 introduces the structure of entries and templates, whilst Section 2.2 formalizes systems and the matching rule processes can use to find entries.

### *2.1 Entries and templates*

Linda tuples are ordered and finite sequences of typed fields. Fields can be *actual* or *formal* (see [CA95]): a field is actual if it specifies the type and a value, whilst it is formal if the type only is given. `SecSpaces` extends this tuple by adding special control fields, namely partition and asymmetric partition fields. Further, for the sake of simplicity, fields are not typed: we consider this aspect does not influence the issues we tackle.

In this section we just want focus the attention on the differences between the definition of entry and template structures. In the next section we will define the matching rule, then we will explain the meaning and the applications of control fields. Control fields are exploited for implementing access control policies based on the kind of operation an agent performs on each entry. Therefore, entries have two occurrences of control fields which are respectively associated to the *rd* (read) and *in* (input) operations, whilst templates have only one occurrence of control fields without any reference to specific primitives. Similarly to SecOS, we test the right access permission of the agent to perform an operation in the matching rule. Hence, the matching rule will evaluate only those entries control fields associated to the operation the agent is performing.

Formally, let $Mess$, ranged over by $m$, $n$, ..., be an infinite set of messages,

*Partition*, ranged over by $c$, $c_t$, $c_s$, ..., be the set of partitions. We also assume that *Partition* contains a special default value, say #, whose meaning will be discussed in the following. Let *APartition*, ranged over by $k$, $k'$, $k_t$, ..., be the set of asymmetric partitions. Similarly to partition fields, we assume that *APartition* contains a special default value denoted by "?" as well. Let "$\overline{\cdot}$ : *APartition* $\rightarrow$ *APartition*" be a function such that if $\overline{k} = k'$ then $\overline{k'} = k$ (and $\overline{?} = ?$). Let *Var*, ranged over by $x$, $y$, ..., be the set of data variables. We use $\vec{x}$, $\vec{y}$, ..., to denote finite sequences $x_1; x_2; \ldots; x_n$ of data variables.

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

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

where the tuple of data $\vec{d}$ is defined by the following grammar:

$$\vec{d} ::= d \mid d; \vec{d}$$
$$d ::= m \mid c \mid k \mid x$$

and $c, c' \in Partition$, $k, k' \in APartition$.

A *data field* $d$ can be a message, a partition, an asymmetric partition or a variable. In the following, we use *Data* to denote the set of data fields. We define $\tilde{e}$ as the operator that, given an entry $e$, returns its tuple of data (e.g., if $e =< \vec{d} >^{[c]_{rd}[c']_{in}}_{[r]_{rd}[r']_{in}}$, $\tilde{e} = \vec{d}$).

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

$$t =< \vec{dt} >^{[c_t]}_{[k_t]}$$

where $\vec{dt}$ is defined by the following grammar:

$$\vec{dt} ::= dt \mid dt; \vec{dt}$$
$$dt ::= d \mid null$$

and $c_t \in Partition$, $k_t \in APartition$.

Differently from entries, data fields of templates can be set to wildcard value, denoted by *null*. Wildcards are used to match with all fields values, e.g., they have the same meaning of formal fields in Linda or the null pointer in JavaSpaces.

For the sake of simplicity, when the partition or the asymmetric partition fields of entries/templates are set to default values, we omit to represent them (e.g., instead of $t =< \vec{dt} >^{[\#]}_{[k]}$ we write $t =< \vec{dt} >_{[k]}$) whilst -in the case of entries- if their value for $rd$ and $in$ is the same we simply write one occurrence without specifying the operation (e.g., instead of $e =< \vec{d} >^{[c]_{rd}[c]_{in}}$ we simply write $e =< \vec{d} >^{[c]}$).

### *2.2 Programs*

Here we formalize systems configurations, that include process exploiting `SecSpaces` coordination primitives and the state of the shared tuple space. Systems are modeled following the approach adopted in [BGZ98] to model Linda primitives: systems are terms obtained by the parallel composition of processes and entries stored into the space.

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 |

| | | |
|---|---|---|
| $P, Q, \ldots ::=$ | 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 |

A system can be an entry, a process or the parallel composition of entries and processes. A process can be a terminated program ($\mathbf{0}$), a prefix form $\mu.P$, the parallel composition of two programs, or the replication of a program. The prefix $\mu$ 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 $P \mid Q$ that is the parallel composition of two processes $P$ and $Q$ behaves as two processes running in parallel. Finally, a process can use the replication operator $!P$, whose meaning is the parallel composition of infinite copies of $P$.

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 $rd/in\ < \vec{dt} >^{[c]_{rd}[c']_{in}}_{[k]_{rd}[k']_{in}} (\vec{x})$ 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.

As previously mentioned, the matching rule between entries and templates depends on the operation the process is performing on the TS. More precisely, there are two access permissions, one associated to $rd$ operations and the other one to $in$ operations. The definition of the matching rule follows.

**Definition 2.1 Matching rule -** Let $e =< d_1; d_2; \ldots; d_n >^{[c]_{rd}[c']_{in}}_{[k]_{rd}[k']_{in}}$ be an entry and $op \in \{rd, in\}$ be the operation using the template $t =< dt_1; dt_2; \ldots; dt_m >^{[c_t]}_{[k_t]}$. Let $c_e$ and $k_e$ be the control fields of $e$ associated to $op$, we define $e$ $matches_{op}$ $t$ as follows:

(i) $m = n$

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

(iii) $c_e = c_t$

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

Condition 1. checks that $e$ and $t$ have the same number of data fields, condition 2. regards the tests applied on data fields and, substantially, it rephrases the matching rule of Linda. Tuples of data fields match if each data of the template is equal to the corresponding data of the entry or if it is set to wildcard value. Condition 3. tests that the partition field of the entry is equal to that of the template. Condition 4. checks that the asymmetric partition field of the template corresponds to the co-key associated to the asymmetric partition field of the entry.

As pointed out by the matching rule, the aim of partition fields is to provide a special kind of data field that does not accept wildcard. Hence, each process that is able to use a template matching with a specific entry must know the partition field of that entry. Partition fields logically partitionate the TS and the access to a partition is restricted to only those processes that know the partition identifier. However, the default value # of the partition field that all agents know (i.e., we assume it to be in the knowledge of any agent) allows any process to interact with each other using that partition. Similarly, a default value known by any process has also been defined for asymmetric partition fields (denoted by "?").

Differently from partitions, the asymmetric partition fields allow for a way to discriminate the permission of write, read and remove an entry. For example, to read an entry with asymmetric partition field set to $k$ the process must set to $\overline{k}$ the asymmetric partition field of the template, that can be an unknown value to the producer of the entry (in order to write he needs just $k$). Therefore, following the same idea of partitions, properly distributing these values we can assign processes the permission to perform only a subset of possible operations on that entry. Finally, as it will be clear in the following, the return value of read/input operations does not include the control fields, only data stored inside the tuple. Therefore, new access permissions can be

acquired only by performing read/input operations of entries containing partition or asymmetric partition values inside tuples of data.

## 3 Operational semantics

We present the operational semantics of systems by defining the *structural congruence* over systems and by mapping them onto *labeled transition systems* (LTSes). Table 1 describes a relation 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) and, finally, replication operator $!P$ corresponds to an infinite parallel composition of $P$ (vii). The structural congruence over systems is defined as the smallest congruence satisfying rules (i), ..., (viii).

| | | | |
|---|---|---|---|
| (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) | $!P \equiv P \mid !P$ |

Table 1
Structural equivalence.

Formally, let $Act = \{\tau\} \cup \{\overline{e} \mid e \in Entry\} \cup \{t \mid t \in Template\} \cup \{\underline{t} \mid t \in Template\}$, ranged over by $\alpha$, be the set of possible actions: $\tau$ represents an internal action, $\overline{e}$ the output of the entry $e$, whilst $t$ and $\underline{t}$ represent a *in* and a *rd* operation using template $t$, respectively. Let $RetV = \{\tilde{e} \mid e \in Entry\} \cup \{-\}$, ranged over by $\beta$, be the set of possible return values of the operations. More precisely, $-$ is returned by output operations and denotes that there is no return value, whilst the other values can be returned by $in/rd$ operations.

The labeled transition system we use is a quadruple $(System, Act, RetV, \rightarrow)$ where $\rightarrow \subseteq System \times Act \times RetV \times System$. $(A, \alpha, \beta, A') \in \rightarrow$ (also denoted as $A \xrightarrow[\beta]{\alpha} A'$) means that the system $A$ can execute action $\alpha$ with return

$$(1)\ out\ e.P \xrightarrow[-]{\tau} e|P \qquad (2)\ in\ t(\vec{x}).P \xrightarrow[\tilde{e}]{t} P[\tilde{e}/\vec{x}]$$

$$(3)\ rd\ t(\vec{x}).P \xrightarrow[\tilde{e}]{\underline{t}} P[\tilde{e}/\vec{x}] \qquad (4)\ e \xrightarrow[-]{\overline{e}} \mathbf{0}$$

$$(5)\ \frac{A \xrightarrow[-]{\overline{e}} A' \quad B \xrightarrow[\tilde{e}]{t} B' \quad e\ matches_{in}\ t}{A|B \xrightarrow[-]{\tau} A'|B'}$$

$$(6)\ \frac{A \xrightarrow[-]{\overline{e}} A' \quad B \xrightarrow[\tilde{e}]{\underline{t}} B' \quad e\ matches_{rd}\ t}{A|B \xrightarrow[-]{\tau} A|B'}$$

$$(7)\ \frac{A \xrightarrow[\beta]{\alpha} A'}{A \mid B \xrightarrow[\beta]{\alpha} A' \mid B}$$

$$(8)\ \frac{A \equiv B \quad B \xrightarrow[\beta]{\alpha} B' \quad A' \equiv B'}{A \xrightarrow[\beta]{\alpha} A'}$$

Table 2
Operational semantics.

value $\beta$ and it evolves in the system $A'$. Table 2 depicts rules that define the operational semantics of `SecSpaces`; relation $\rightarrow$ is the smallest one satisfying rules $(1), \ldots, (7)$. Rules (1), (2) and (3) describe the three prefix operators *out*, *in* and *rd*, respectively. More precisely, $out(e).P$ performs an internal action (i.e. it is not observable) that produces an occurrence of the entry $e$ in the TS (represented as a parallel component of the system) and then it behaves as $P$; $in\ t(\vec{x}).P$ and $rd\ t(\vec{x}).P$ perform an input and read operation, respectively: when a matching entry $e$ is found in the TS the return value is $\tilde{e}$ and the process behaves as $P[\tilde{e}/\vec{x}]$. The consumption of an entry $e$ in the TS is represented in (4), whilst rule (5) describes that if a process performs an $in\ t(\vec{x})$ and a matching entry $e$ is available in the TS, then the entry is consumed and the input is performed; in this case the system evolves in an internal action. Rule (6) describes read operations and, differently from (5), in this case the matching entry $e$ continues to be stored in the TS. Finally, (7) describes the behaviour of processes running in parallel, whils (8) says that we can replace at any time a system with another one structurally congruent.

In the following, $A \xrightarrow{\tau} {}^* A'$ denotes that the system may evolve to $A'$ performing only $\tau$ moves; that is $A \xrightarrow[-]{\tau} A_1 \ldots \xrightarrow[-]{\tau} A_{n-1} \xrightarrow[-]{\tau} A'$ for some $n$.

## 4 Testing equivalence

In order to compare the behaviour of systems, in this section we rephrase in this context a notion of observational equivalence, namely *may testing equivalence* [NH84].

Informally, two systems are testing equivalent if, whatever is the environment in which the systems are executed, an event occurs in one system if and only if it occurs in the other one. The set of external observers that we consider is parametrized by the set of data the external environment knows. It is worth noting that this precaution in other calculi (e.g. the spi calculus) in which testing equivalence has been defined is not necessary. The problem is that in our model we cannot restrict the scope of a name so that it is bound inside the system, hence the only way we have to assume that a data is secret (i.e. bound to the system) is to exclude this data from the knowledge of the processes in the external environment. The approach we have followed has already been proposed in CryptoSpa [FGM]. Therefore, the testing equivalence we are going to define is parametrized by the set of data known by the external environment. On the contrary, in the spi calculus [AG99], the $\nu$ operator can be used to describe bound data. In this way, the set of external observers is not in the scope of names bound inside the system and such names are ensured to be secret by the binding mechanism itself.

Formally, let $\omega \in Data$ be a *barb*, i.e. any data that we use to detect the success of a test. We say that a system $A$ *immediately exhibits* barb $\omega$ (denoted by $A \downarrow \omega$) iff $A \equiv < d_1; d_2; \ldots; d_n >^{[\#]}_{[?]} |B$, where $\omega = d_i$ for some $i$, $1 \leq i \leq n$, whilst we say that a system $A$ *exhibits* barb $\omega$ (denoted by $A \Downarrow \omega$) iff $A \xrightarrow{\tau} {}^* A'$ and $A' \downarrow \omega$. We also define a *test* to be a couple $(T, \omega)$ where $T$ is a system and $\omega$ is a barb; we say that a system $A$ *immediately passes* a test $(T, \omega)$ iff $A|T \downarrow \omega$, whilst a system $A$ *passes* a test $(T, \omega)$ iff $A|T \Downarrow \omega$. Let $\phi_E \subseteq Data$ be the knowledge set of the external environment, the set $E(\phi_E) = \{A \mid fn(A) \subseteq \phi_E\}$ represents the external environment (set of tests) we consider in the equivalence evaluation.

**Definition 4.1 Testing equivalence -** Let $A$ and $B$ be two systems and $\phi_E$ be the knowledge of the external environment; we say that $A \leq_{may} B$ iff for every test $(T, \omega)$ with $T \in E(\phi_E)$: $(A \mid T) \Downarrow \omega$ implies $(B \mid T) \Downarrow \omega$. $A$ and $B$ are testing equivalent (denoted by $A \approx B$) iff $A \leq_{may} B$ and $B \leq_{may} A$.

It is trivial to check that $\approx$ is an equivalence relation and that systems structurally equivalent are testing equivalent.

In the original work [NH84] a barb is a special action $\omega$ used to signal the success of an experiment. More in general, the barb are the actions considered interesting from the point of view of an external observer. Therefore, in our context, it is reasonable to consider as a barb only data stored inside one of the entries of the default partition (see the usage of "#" and "?" in the definition of "$\downarrow$") because each process can read such data.

# 5 Security properties

In this section we intend to formalize in `SecSpaces` some of the main security properties and describe how some forms of secure coordination can be implemented in `SecSpaces`. Previous sections will be exploited to formally describe interaction protocols and security properties. More precisely, security properties definition exploits testing equivalence.

## *5.1 Data secrecy using partition fields*

The first lack of Linda we have emphasized in the Introduction section is that there is no way to keep secret any data written in the TS. Therefore, the first goal we tackle is to implement a data exchange between two processes that ensures data secrecy. Informally, given a system we say that the secrecy of a datum holds if, whatever is the environment in which the system runs, hostile processes have no way to known that datum.

To formalize this property we follow the same idea used in [AG99]: secrecy of a datum $d$ in a system $S$ holds if the system $S$ is observationally equivalent to $S[d'/d]$, for any $d'$. Intuitively, this means that from the point of view of the external observer two instances of a system which exchange different data are indistinguishable, i.e. it cannot distinguish which is the exchanged data. Formally, we say that the secrecy property -of a datum $d$- holds in the system $S$ iff $S \approx S[d'/d]$ for any $d'$. It is worth noting that exchanged data can be also known by the hostile environment, but they have no way to understand which value has been exchanged.

In order to obtain a form of interaction in which secrecy holds, we need just to use partition control fields. The idea we follow is that if a partition field, say $c$, is known only by two agents and if they keep it secret, they can exchange any data ensuring that secrecy holds simply by using entries with partition field set to $c$. Formally, let $A(d) = out(< d >^{[c]}).\mathbf{0}$ be the process that writes the entry $< d >^{[c]}$ and then terminates, $B = in\ < null >^{[c]}\ (x).\mathbf{0}$ be the process that removes an entry having partition field set to $c$. Let $S(d) = A(d)|B$ be a system; if only $A$ and $B$ know $c$, i.e. $c \notin \phi_E$, the exchange of the data $d$ between $A$ and $B$ satisfies the secrecy property, that is: $S(d) \approx S(d')$ for any $d$ and $d'$. It is rather easy to prove that this condition holds for the system we have defined. Indeed, as an intuition, during the computation $c$ will never be in the hostile environment knowledge and, consequently, because hostile environment cannot access to the entries of partition $c$, it cannot distinguish which data the protocol exchanges.

Note that, as we previously said in the Introduction section, by just using partition control fields, we cannot discriminate between the read and the write permission of an entry. In the following we will see properties where this discrimination of access permissions is needed and we exploit asymmetric partition control fields.

### *5.2 Producer and receiver authentication*

`SecSpaces` access control mechanisms are finer enough to assign at the agents the permission to perform a subset of operations on a specific entry. In this section we consider a special case: only one agent, say A, has write permission on a certain class of entries. In this way, each entry (of this class) stored in the TS has been written by $A$ and we say that the property of *producer authentication* holds. Informally, producer authentication of a specific entry holds if, whatever is the hostile environment in which the system runs, that entry can be generated only by the specified producer.

As announced in the introduction, we exploit the different knowledge an agent must have in order either to read (remove) an entry or to write that entry when asymmetric partition fields are used. Let $P(d) = out(< d >_{[k]}).\mathbf{0}$ be the process that writes the entry $< d >_{[k]}$ into the TS, and $R = in \ < null >_{[\overline{k}]} (x).F(x)$ be the process that reads an entry (with asymmetric partition field set to $k$) and then it continues with $F$ using the received data. If only $P$ knows $k$, i.e. $k \notin fn(R)$ and $k \notin \phi_E$, and it keeps $k$ secret, $P$ is the only process with write permission on the class of entries having $k$ as asymmetric partition field. Consequently, the system $S(d) = P(d)|R$ is such that the property of producer authentication -of the entry written by $P$- holds. In order to prove this property we proceed as follows: i) we define the ideal behaviour of the system: $S_{id}(d) = P(d)|R_{id}(d)$, where $R_{id}(d) = in \ < d >_{[\overline{k}]} (x).F(d)$; ii) we test if the ideal system $S_{id}(d)$ is testing equivalent to the system $S(d)$, for any $d$. Intuitively, the ideal system is such that the received entry is certainly that written by $P$ (because $R_{id}$ admits only exact matching) whilst the original system is such that it reads any entry with one data field and asymmetric partition field set to $k$ and then the continuation depends on the data it reads. If the two systems are testing equivalent for any $d$, it means that the external environment cannot produce any entry having asymmetric partition field set to $k$. Note that we have made no assumption on $\overline{k}$, i.e. the external environment can read (remove) the entry written by $P$.

It is easy to prove that the system satisfies producer authentication, that is $S_{id}(d) \approx S(d)$, for any $d$. It is worth noting that the idea we exploit can be used in a similar way to assign write permission to a specific set of agents simply by allowing the knowledge of $k$ only to those agents.

Symmetrically, by limiting only to $R$ the knowledge of $\overline{k}$ we obtain the property of *receiver authentication* (of an entry); receiver authentication holds if, whatever is the external environment, $R$ is the only process that can read a specific entry. Obviously, R can read the entry P writes and, in general, can read any entry (with one data field) having asymmetric partition field set to $k$. In order to prove that receiver authentication holds, i.e. that only $R$ can read such entries, we proceed as follows: we show that different entries with asymmetric partition field set to $k$ are indistinguishable from the point of view of an external observer, i.e. it cannot access those entries. In this case,

receiver authentication holds iff $P(d) \approx P(d')$, for any $d$ and $d'$. It is trivial to prove that this condition is satisfied if we assume $\overline{k} \notin \phi_E$.

### *5.3 Message authentication*

Several applications need to ensure that an agent reads exactly a specific data. For instance, let us consider the case in which an agent wants to receive the IP address of a trusted machine; it can be useful to guarantee that the received IP address is exactly the expected one. This security property is usually referred to as *message authentication* [FGM00,AG99].

To formalize this property we proceed in the same manner used for producer authentication: i) we define the ideal system satisfying (by construction) message authentication; ii) we test if the ideal system is testing equivalent to the system.

In the previous section we have shown how to satisfy producer authentication; the idea we follow is that message authentication can be satisfied by exchanging data using entries whose producer can be authenticated (it is, e.g., a trusted server). Let S(d) be the system we have defined in the previous section, then -in the case in which $k$ is known only by $P$- the system satisfies message authentication of the data $d$. Let $S_{id}(d) = P(d)|in\ < null >_{[\overline{k}]}\ (x).F(d)$ be the ideal system in which the receiver continues using the expected data $d$; message authentication holds iff $S_{id}(d) \approx S(d)$, for any $d$. It is easy to test this condition holds for the given system.

### *5.4 A simple protocol for secrecy and authentication*

The protocol proposed in Section 5.1 exploits the partition field $c$ -shared only by agents $A$ and $B$- to exchange data (on that partition) satisfying secrecy property. In that example we have assumed $c$ is in the initial knowledge of $A$ and $B$; here we describe how to exchange $c$ by exploiting a protocol that guarantees secrecy of $c$ and producer, receiver and message authentication (Sections 5.2 and 5.3). We assume that: i) $c$ is the partition $A$ wants to communicate to $B$ in order to use that partition to exchange data in a secure way; ii) only $A$ knows $k$; iii ) only $B$ knows $\overline{k}$. Let $A = out(< c >_{[k]}).\ in\ < null >^{[c]}\ (x).\mathbf{0}$ be the process that writes the entry $< c >_k$ and then performs an input of an entry having partition field set to $c$, $B(d) = in\ < null >_{[\overline{k}]}\ (x).out(< d >^{[x]}).\mathbf{0}$ be the process that performs an input of an entry with asymmetric partition field set to $k$ and then writes an entry in the partition $x$ returned from the input. Let $\phi_E$, with $c, k, \overline{k} \notin \phi_E$, be the knowledge of the hostile environment; the system $S(d) = A|B(d)$ ensures the secrecy of the data exchanged in the partition field $A$ communicates to $B$ (i.e., $c$): $S(d) \approx S(d')$ for any $d$ and $d'$. Moreover, it is trivial to prove producer and receiver authentication of the entry $< c >_{[k]}$ holds. Therefore, the protocol guarantees mutual authentication: $A$ and $B$ cannot repudiate to have performed the protocol.

### *5.5 Discriminating read/remove permissions*

Until now we have used systems that communicate using control fields having the same value for both read and remove operations ($rd$ and $in$). Obviously, all the methodologies proposed can be applied -using control fields with different values for $rd$ and $in$- in order to assign to an agent only read or remove permission. One interesting advantage is that make it possible to ensure the *availability* of a datum. Indeed, exploiting the $rd$ operation of Linda an entry in TS can be read by more than one process. Let us consider the case in which a process is willing to communicate some data -contained into an entry- to a group of agents; in the case $rd$ and $in$ access permissions coincide a reader can maliciously remove the entry that becomes unavailable to the other processes. On the other hand, exploiting different access permissions we can easily ensure data availability by allowing -to the specified group of processes- only read access permission to the entry. Finally, when we assign only $in$ access permission to an entry, it can be read only by one process. More precisely, we say that an entry has *remove-only* permission if it can be accessed only performing $in$ operations, then only one process can access that entry; this can be useful in those applications that need to distribute a set of data to a set of processes guaranteeing that the data are read only by one process (e.g., a process that collects service requests and then distribute them to a set of processes that perform the requested jobs).

As an interesting case, here we illustrate the case in which the producer of an entry assigns it *read-only* access permission (the same technique we use can be exploited to assign remove-only access permission). In other words, this means that no agent (except the owner) can remove the written entry (i.e. perform an $in$). This application can be useful in several contexts; for example broadcast communications or in those applications which have to publish personal information to each other (e.g., ip-address, phone-number).

Informally, we can say that an entry $e$ can be accessed only using read operations if whatever is the hostile environment $T$, the parallel composition of $e$ and $T$ always evolves into a configuration in which the entry $e$ is still in the space. Formally, let $e$ be an entry and $\phi_E$ be the knowledge of the hostile environment; we say that $e$ has read-only permission iff for any $T \in E(\phi_E)$ : $e \mid T \rightarrow^* A$ implies $A \equiv e \mid A'$.

In order to assign the read-only permission to an entry we proceed as follows: we assign to the partition (or to the asymmetric partition) control field corresponding to the "$in$" operation a value that is known only to the producer. In this way, no process can perform an input because it does not have the necessary knowledge. As a simple example, let $A = out(< d >^{[c]_{rd}[c']_{in}}).P$ be the process that writes the entry $< d >^{[c]_{rd}[c']_{in}}$ and then behaves as $P$; if only $A$ knows $c'$ (i.e., $c' \notin \phi_E$) and it keeps $c'$ secret (i.e., secrecy of $c'$ holds in $P$) the written entry has read-only permission. It is worth noting that we can allow read-only access permission to a restricted set of processes by limiting

the knowledge of $c$ to that set of processes.

## 6 Related work and conclusion

In this paper we have proposed a process algebra for `SecSpaces` and rephrased in this context the classical notion of may testing equivalence. Moreover, we have shown that by using such a formal machinery we can express security properties (such as secrecy and authentication) and check system algebraic specifications against them. Finally, we have presented some examples of property checking for some simple interaction protocols. Such examples have given evidence that our approach provides granularity of access permission at the level of the kind of operation performed on a tuple.

Most inherent previous proposals have already been commented. Here we add only a comment about proposals based on the orthogonal approach in which secure coordination is based on mechanisms that ensure privacy of the data stored inside tuples (e.g., cryptography). `SecSpaces` can be easily extended to support this feature by exploiting cryptography similarly as in [BN02]: we could extend the syntax of data field by adding encrypted data, i.e. $d ::= \ldots \mid \{d\}_{E_c}$, where $\{d\}_{E_c}$ denotes the encryption of $d$ with the cryptographic key $E_c$. In this way, the content of a data field encrypted can be accessed only by those processes that know the corresponding decryption key.

In this work we were able to formalize some security properties, namely safety properties only, by using may testing equivalence (see [NH84] for more details). In the literature there are several other approaches used to formalize and to verify security properties in process calculi, e.g. type systems [NFP98,GP03] or proof systems [Mar98]. As future work we intend to formalize a notion of non-interference, originally presented in [GM82] and then developed in many works (see, for example, [FGM00]), in order to be able to capture more properties (e.g., the information flow security properties of a multilevel systems).

## References

[AG99] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[BGLZ02] Nadia Busi, Roberto Gorrieri, Roberto Lucchi, and Gianluigi Zavattaro. Secspaces: a data-driven coordination model for environments open to untrusted agents. In *1st International Workshop on Foundations of Coordination Languages and Software Architectures*, volume 68.3 of *ENTCS*, 2002.

[BGZ98] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer*

*Science*, 192(2):167–199, 1998.

[BN02] L. Bettini and R. De Nicola. A Java Middleware for Guaranteeing Privacy of Distributed Tuple Spaces. In *Proc. of FIDJI'02, Int. Workshop on scientific engineering of distributed Java applications*, 2002. To appear in LNCS.

[CA95] Scientific Computing Associates. *Linda: User's guide and reference manual*. Scientific Computing Associates, 1995.

[CG92] N. Carriero and D. Gelernter. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–106, 1992.

[FGM] R. Focardi, R. Gorrieri, and F. Martinelli. Message Authentication through Non Interference. In Proc. of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST '00), Lecture Notes in Computer Science 1816, Springer-Verlag, pp. 258-272, Iowa City (USA), May 2000.

[FGM00] R. Focardi, R. Gorrieri, and F. Martinelli. Non interference for the analysis of cryptographic protocols. In U.Montanari and E.Welzl, editors, *International Colloquium on Automata, Languages and Programming (ICALP'00)*, volume 1853 of *LNCS*. Springer, July 2000.

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

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GM82] J.A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Comp. Soc. Press., 1982.

[GP03] Daniele Gorla and Rosario Pugliese. Resource Access and Mobility Control with Dynamic Privileges Acquisition. In *Proc. of the Int. Colloquium on Automata, Languages and Programming (ICALP'03)*, LNCS, 2003. To appear.

[Mar98] Fabio Martinelli. Partial Model Checking and Theorem Proving for Ensuring Security Properties. In *Proc. of the IEEE Computer Security Foundations Workshop (CSFW'98)*, pages 44–52. IEEE Comp. Soc. Press., 1998.

[MMU01] N. Minsky, Y. Minsky, and V. Ungureanu. Safe Tuplespace-Based Coordination in Multi Agent Systems. *Journal of Applied Artificial Intelligence*, 15(1), 2001.

[NFP97] R. De Nicola, G. Ferrari, and R. Pugliese. Coordinating Mobile Agents via Blackboards and Access Rights. *In Proc. of the Second International Conference on Coordination Models and Languages, Lectures Notes in Computer Science 1282, Springer*, pages 220–237, 1997.

[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.

[NH84] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[Pin92] J. Pinakis. Providing directed communication in Linda. *In Proceedings of the 15th Australian Computer Science Conference*, pages 731–743, 1992.

[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.