Security, Probability and Priority in the tuple-space Coordination Model

Roberto Lucchi

March 2004

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)
Recent Titles from the UBLCS Technical Report Series


2003-16 Robust Aggregation Protocols for Large-Scale Overlay Networks, Montresor, A., Jelasity, M., Babaoglu, O., December 2003.


Security, Probability and Priority in the tuple-space Coordination Model

Roberto Lucchi

March 2004
Abstract

New networking technologies are moving to support applications for open systems (e.g., peer-to-peer, ad-hoc networks, Web services), in which the entities that will be involved in the application are unknown at design time. Further, the connectivity is exploding: a growing number of devices need to communicate with each other and the new challenge is how to design and to program the communication among devices.

Coordination models and languages, which advocate a distinct separation between the internal behaviour of the entities and their interaction, represent a promising approach for the development of this class of applications. The interaction is programmed according to a coordination infrastructure that abstracts away from the exact name/location of the components and the underlying network. One of the most prominent coordination languages is Linda in which a shared space, containing tuples of data, is used by agents to collaborate. Agents can insert new tuples into the space, consume or read tuples from the space thus implementing the so-called generative communication, in which tuples are independent of their producers.

We investigate the security issues, from the point of view both of the model and of the implementation, that emerge when Linda is used in open systems where any agent can enter. In this scenario, in which the system may comprise malicious agents, designers have to deal with security. Unfortunately, Linda is not expressive enough to provide security solutions because any agent having
access to the space can read/remove/reproduce any tuple available in the shared space.

We present a novel coordination language, SecSpaces, which follows the tuple-space coordination model introduced by Linda, that supports security in untrusted environments by providing some access control mechanisms on tuples. We also implement the SecSpaces language as a Web service, that represents the emerging networking technology for programming Internet applications.

Besides security mechanisms we also introduce, in a process algebraic setting, a more sophisticated data retrieval mechanism for Linda that supports priority and probability. Priorities are used to express an absolute preference: among the tuples that could be accessed in the classical data-retrieval primitives of Linda, only those with highest priority can be selected. Probabilities allow for expressing the probability to access a specific tuple among those that could be retrieved.

Finally, we show that the access control mechanisms of SecSpaces and the probabilistic/prioritized data-retrieval mechanisms we propose on tuples can be combined. In particular, we describe how probabilities and the SecSpaces access control mechanism can be successfully exploited to implement a protocol for the registration and the discovery of Web services.
Acknowledgements

This Ph.D. thesis is the final result of three years of graduate studies in which I had the possibility to collaborate and discuss with several people that stimulated my research activity, and to attend several meetings and advanced schools.

First of all I would like to thank my supervisor Prof. Roberto Gorrieri that, with his experience in coordination models and security research fields, gave me very valuable comments and suggestions. I also thank him for the financial support and the travel opportunities he gave me.

A special thank goes to Gianluigi Zavattaro and Mario Bravetti that gave a significant contribute to my research. During these years, Gianluigi collaborated with me to the definition of an extension of the Linda coordination language supporting security, probabilistic and prioritized mechanisms, giving me helpful suggestions and comments. Mario, with his precious experience in probabilistic and stochastic process algebras, gave me the possibility to investigate how probability and priority can be used to extend the tuple space coordination model.

Many other people in Bologna have been precious during these Ph.D. years, among all I would like to mention Nadia Busi and Riccardo Focardi that gave me interesting feedback about my research activity.

I would like to mention also the reviewers, G.Catalin Roman and Jan Vitek, whose observations have been very helpful during the final revision of this document.
Contents

Abstract vii

Acknowledgements ix

List of Figures xiv

1 Introduction 1
  1.1 Introduction ................................................. 1
  1.2 The Linda coordination model ............................... 2
  1.3 Linda for Open Systems: security issues .................... 3
  1.4 Linda-like languages for untrusted environments .......... 5
    1.4.1 KLAIM .................................................. 6
    1.4.2 SecOS ................................................... 7
    1.4.3 Other proposals ......................................... 8
  1.5 Aims and Scope of the thesis ............................... 9
  1.6 Related publications ...................................... 12

2 Security in tuple space coordination model 15
  2.1 Introduction ............................................... 15
  2.2 Security Properties in Linda ............................... 20
    2.2.1 Modeling the coordination medium ...................... 20
    2.2.2 Modeling the agents .................................... 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.5</td>
<td>3.5.5 Discriminating read/remove permissions</td>
<td>71</td>
</tr>
<tr>
<td>3.6</td>
<td>3.6 Conclusion</td>
<td>73</td>
</tr>
<tr>
<td>4</td>
<td>4 Supporting Web Services Coordination with SecSpaces</td>
<td>75</td>
</tr>
<tr>
<td>4.1</td>
<td>4.1 Introduction</td>
<td>75</td>
</tr>
<tr>
<td>4.2</td>
<td>4.2 The WSSecSpaces Web Service</td>
<td>79</td>
</tr>
<tr>
<td>4.2.1</td>
<td>4.2.1 Implementation of control fields</td>
<td>80</td>
</tr>
<tr>
<td>4.2.2</td>
<td>4.2.2 Entry, template and XML-based matching rule</td>
<td>81</td>
</tr>
<tr>
<td>4.2.3</td>
<td>4.2.3 WSDL and implementation</td>
<td>85</td>
</tr>
<tr>
<td>4.3</td>
<td>4.3 Related work</td>
<td>86</td>
</tr>
<tr>
<td>4.4</td>
<td>4.4 Conclusion</td>
<td>87</td>
</tr>
<tr>
<td>5</td>
<td>5 Combining partitions in SecSpaces</td>
<td>89</td>
</tr>
<tr>
<td>5.1</td>
<td>5.1 Introduction</td>
<td>89</td>
</tr>
<tr>
<td>5.2</td>
<td>5.2 SecSpaces with combined partitions</td>
<td>91</td>
</tr>
<tr>
<td>5.2.1</td>
<td>5.2.1 The language</td>
<td>91</td>
</tr>
<tr>
<td>5.2.2</td>
<td>5.2.2 Semantics</td>
<td>95</td>
</tr>
<tr>
<td>5.3</td>
<td>5.3 Group communication examples</td>
<td>97</td>
</tr>
<tr>
<td>5.3.1</td>
<td>5.3.1 Group restriction</td>
<td>99</td>
</tr>
<tr>
<td>5.3.2</td>
<td>5.3.2 Group coordination</td>
<td>100</td>
</tr>
<tr>
<td>5.4</td>
<td>5.4 A system with multi-level security</td>
<td>101</td>
</tr>
<tr>
<td>5.5</td>
<td>5.5 Conclusion</td>
<td>104</td>
</tr>
<tr>
<td>6</td>
<td>6 Probabilistic and Prioritized Data Retrieval in Linda</td>
<td>105</td>
</tr>
<tr>
<td>6.1</td>
<td>6.1 Introduction</td>
<td>105</td>
</tr>
<tr>
<td>6.2</td>
<td>6.2 The Linda coordination model</td>
<td>109</td>
</tr>
<tr>
<td>6.3</td>
<td>6.3 Adding probabilistic data-retrieval to Linda</td>
<td>113</td>
</tr>
<tr>
<td>6.3.1</td>
<td>6.3.1 Probabilistic model</td>
<td>113</td>
</tr>
<tr>
<td>6.3.2</td>
<td>6.3.2 Syntax</td>
<td>115</td>
</tr>
</tbody>
</table>
6.3.3 Semantics .................................................. 117
6.4 Adding prioritized data-retrieval to Linda ...................... 121
   6.4.1 Static priorities ........................................ 121
   6.4.2 Dynamic name-based priorities ......................... 124
6.5 Conclusion and related work .................................. 127

7 Priorities and Probabilities in SecSpaces ...................... 129
   7.1 Introduction ............................................. 129
   7.2 Priorities in the SecSpaces model ......................... 131
   7.3 Probabilities in the SecSpaces model ...................... 134
      7.3.1 Implementation ...................................... 136
   7.4 Supporting Web services workload distribution with probabilistic SecSpaces ..................... 137
   7.5 A Web service registry guaranteeing Security and Quality of Service 140
      7.5.1 Web service discovery .............................. 140
      7.5.2 Registration and deregistration of Web services ........ 143
      7.5.3 An Example of Application: A Software Factory ......... 144
   7.6 Conclusion ................................................ 146

A WSSecSpaces specifications ................................... 147
   A.1 Entry and templates XML-schema .......................... 147
   A.2 WSDL of WSSecSpaces ................................. 149
   A.3 Entry with weight ........................................ 154

References ...................................................... 157
List of Figures

1.1 Linda architecture .................................................. 4

4.1 Web service technologies ............................................. 77

4.2 The role of Coordination languages in Web service technologies . 78
Chapter 1

Introduction

1.1 Introduction

New networking technologies are calling for the definition of models and languages adequate for the design and management of new classes of applications. Innovations are moving towards two directions. On the one hand, the Internet is the candidate environment for supporting the so-called wide area applications. On the other hand, smaller networks of mobile and portable devices, such as mobile ad-hoc networks or peer-to-peer systems, support applications based on agents or components which interact according to a dynamically reconfigurable communication structure. In both cases, the challenge is to develop applications without knowing, at design time, the overall structure of the system as well as the entities that will be involved. Indeed, these systems are usually open to new agents or components which are unknown beforehand.

Furthermore, our society is becoming more and more dependent on computer networks: the enormous amount of data that is elaborated, transmitted or stored needs some forms of protection. Hence, in order to guarantee some security properties, several procedures based on cryptography (see, e.g. [Sch96]) have been proposed in the literature. The goals of these protocols cover a large area of
applications, e.g. privacy and authentication of the messages, personal identification, digital signatures, electronic money transfer, credit card transactions and many other critical applications. Actually, security protocols and the applications exploiting them are typically based according to a channel based topology.

Coordination models and languages, which advocate a distinct separation between the internal behaviour of the entities and their interaction, represent a promising approach for the development of applications for this class of dynamic and open systems. For instance, we assist to a renewed interest in data-driven coordination infrastructures originated by Linda [Gel85] as exemplified by recent commercial products, such as JavaSpaces [Sun02] and TSpaces [WMF98], which are two coordination middlewares for distributed Java [GJS96] programming proposed by Sun Microsystem and IBM, respectively. Both proposals exploit the so-called generative communication [Gel85]: a sender communicates with one or more receivers through a shared tuple space (TS for short), where emitted tuples are collected; a receiver can read or consume the tuples from the TS; a tuple generated by an agent has an independent existence in the tuple space until it is explicitly withdrawn by a receiver; in fact, after its insertion in TS, a tuple becomes equally accessible to all agents, but it is bound to none. This form of communication is referred to as generative communication because when a datum is produced, it has an existence which is independent of its producer, and it is equally accessible to all agents.

1.2 The Linda coordination model

The Linda language provides coordination primitives that allow processes to insert new tuples in the TS or to access the tuples already stored in the share tuple space. More precisely, a tuple is a sequence of typed values [CA95] and a TS is a multiset of tuples.

Processes can exchange tuples through introducing them into the TS. The primitive \( \text{out}(e) \) permits to add a new occurrence of the tuple \( e \) to the TS.
The data-retrieval primitives permit processes, by specifying a template \( t \), to access tuples available in the TS that match the template. More precisely, a template is a sequence of fields that can be either actual or formal: a field is actual if it specifies the type and a value, while it is formal if the type only is given. Two typed values match if they have the same value, while a typed value matches a formal fields if it has the type specified in the latter. A tuple \( e \) matches the template \( t \) if \( t \) and \( e \) have the same arity and each field of \( e \) matches the corresponding field of \( t \).

The \( \text{in}(t) \) is the blocking input primitive: when a tuple \( e \) matching the template \( t \) is available in the TS, an occurrence of \( e \) is removed from the TS and the primitive returns \( e \). The \( \text{rd}(t) \) primitive is the blocking read primitive: differently from the \( \text{in}(t) \), when a tuple \( e \) matching the template \( t \) is in the TS, the primitive returns \( e \) without removing it from the TS.

Linda also provides the non-blocking version of the data-retrieval primitives: the \( \text{inp} \) and the \( \text{rdp} \) are the non-blocking version of the \( \text{in} \) and the \( \text{rd} \), respectively. If the tuple \( e \) is in the TS, their behavior is the same as for the blocking operations, otherwise they return a special value indicating the absence of \( e \) in the TS.

### 1.3 Linda for Open Systems: security issues

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 proposed in Chapter 4 for Web services, Lime [MPR01] in the context of MANETs and PeerSpaces [BMMZ03] for P2P applications) new critical aspects come into play such as the need to deal with a hostile environment which may comprise also untrusted components.

Figure 1.1 describes a basic scheme of the architecture of systems based on the Linda model. The elements represented are Clients, Space-Plug (SP for short) and
the Space Manager. The Space-Plugs are ports that provide clients the interface for using the Linda primitives. Clients use the communication channels client-SP to transmit service requests and to receive return values. One of the most important entry points for an attack are these communication channels. The risk is raised because enemies can eavesdrop or manipulate the exchanged data along the path client-SP, thus altering the expected behavior. Furthermore, an attacker SPE (Space-Plug Enemy) can replace a SP in the interaction with the clients. If this happens, clients read and write information from and to a fraudulent server SPE (that acquires all transmitted data) believing to interact with SP. However, security issues in transmission on channel, such as entity authentication, privacy and integrity of data, are well known problems and several cryptographic protocols [Sch96] have been proposed to solve them. To investigate these aspects is outside the scope of this thesis.

The Space Manager (SM for short) is the entity that manages the operation on the tuple space, according to the semantics of the primitives. Clearly, if we look inside the SM, we can detect several other security issues, such as how to distribute the data or how the Space Plugs interact with SM. In this thesis we do not investigate the security issues related to the design and the implementation of the Space Manager.
Finally, also untrusted agents may enter the system and, according to the data-driven approach, they can access the repository in order to read/remove data, as well as maliciously produce new data. More precisely, each agent, provided it can access a SP, can perform any primitive on the space. In particular (as it will be explained in detail in the following chapter), the main problem is that any agent can read/remove any tuple stored into the TS simply by exploiting formal fields, that acts as wildcards. The security issues we tackle in this thesis are related to this problem, that can be solved in two different manners: i) by providing an authentication mechanisms that SPs can use to allow the access to the Space Manager only to authorized agents, or ii) by implementing an access control mechanism that permits the access to the tuples only to the authorized agents, so that even if malicious agents access the space they cannot read/remove such tuples. We consider that the first methodology is not suitable for supporting open environments because, in order to manage capabilities, the system should know all possible agents having access to the space and this is a strong assumption in this specific context.

1.4 Linda-like languages for untrusted environments

The different solutions that have been proposed in order to support the tuple-space coordination model in open environments can be distinguished into two classes depending on their (orthogonal) goals: either they are based on access control mechanisms [NFP98, VBO03, Pin92, MMU01, BGLZ03], or on mechanisms capable to ensure privacy of exchanged data (e.g., cryptography, [BN03]). As previously explained, each agent having the access to the TS can read/remove each tuple stored in the TS. This means that there is no way to restrict the accesses to a tuple only 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 the TS. The former approach proposes a solution to this problem, while the second one exploits cryptography to allow only
to authorized users the access to the specific information stored inside the tuples, but nothing is done to restrict the access to the tuples stored in the TS.

The most interesting proposals of data-driven coordination languages which provide some forms of access control are KLAIM, described in Section 1.4.1, and SecOS, described in Section 1.4.2. Other related work are described in Section 1.4.3.

1.4.1 KLAIM

KLAIM [NFP98] (Kernel Language for Agent Interaction and Mobility) is designed to program distributed applications that can interact using multiple tuple spaces and mobile code. The KLAIM operations basically are the same of Linda, in addition they allow the reference to a specific tuple space.

The basic element is the location that is composed by a tuple space and by the processes running in that location. In order to specify the tuple space, a reference to a location is given, for instance \( /D3/D9/D8 \), \( /B4/CT/B5 @/D0 \), and \( /D6/CS/B4/D8/B5 @/D0 \) are performed at location \( /D0 \).

The model does not allow dynamic privileges acquisition, then the capabilities are completely defined at design time. Section 1.4.3 reports an extension of the core of KLAIM that supports this feature. In order to express access permissions KLAIM describes which primitives each location (i.e. the processes running in that location) can perform on each of the available locations. For instance, \( \text{out}(e)@l \), \( \text{in}(t)@l \), and \( \text{rd}(t)@l \) are performed at location \( l \).

Types are used to express location capabilities, that is the capabilities associated to the processes running in that location. In order to describe the primitives that processes are willing to perform a type system has been defined. By exploiting these types, a static type-checking has been introduced for verifying whether the access permissions associated to the location where the processes run allow for the execution of the primitives declared by the processes.
1.4.2 SecOS

SecOS [VBO03] is a capability system that supports some forms of control of the accesses to the tuples. In order to express access permissions two kinds of access keys can be used to refer each piece of data: the symmetric and the asymmetric access keys. 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.

The access keys can be associated to either the tuples (object locks) or the single fields contained into tuples (field locks). For instance, the tuple $\langle a_a : 2 \ b_b : 8 \rangle@c_d$ is locked by the asymmetric access key $c_d$ and contains two fields, locked by the symmetric access keys $a_a$ and $b_b$, whose values are 2 and 8, respectively. The selection operator can be used to access single fields by specifying the access key; for instance $\langle a_a : 2 \ b_b : 8 \rangle.a_a$ selects the value of the field locked by $a_a$, that in this case is 2. When a tuple/field is locked with an asymmetric key, say $a_b$, the selection requires the corresponding co-key denoted by $\overline{a_b}$, whose definition is: $\overline{a_b} = b_a$.

The primitives provided by SecOS are the output and the blocking input. For instance, $\text{out} \langle a_a : 2 \rangle@c_d$ produces a tuple locked with access key $c_d$, that can be accessed only by processes that provide $d_c$ as object access key, e.g., in $\langle a_a : ? \rangle@d_c$ (? denotes that the corresponding field of the tuple can contain any value). Data in the tuples can contain access key values, thus the shared space can be used to perform the distribution of access permissions.

Differently from Linda, the matching is also allowed between tuples and templates with different length and order. By considering that the return value of data-retrieval primitives is the matching tuple at least a short discussion is needed. The fact that processes acquire not only the fields they are authorized to access but rather all those stored in the matching tuple does not represent a security risk. Indeed, the data that should be protected are really inaccessible if the process does not possess the corresponding access key. Furthermore, to search a matching tuple in the tuple space is more expensive than in Linda. In order to
improve the performance, the implementation of SecOS that has been proposed uses an efficient inequality test between tuples and templates and a binary search tree.

From the point of view of the implementation, it is worth noting that the same access keys can be exploited, in the implementation, in order to protect [BOV99] the content of the tuples when they are transmitted in the Internet, simply by considering access keys as cryptographic keys.

1.4.3 Other proposals

Other two proposals, muKLAIM [GP03] and Klava [BN03], are extension of the KLAIM model. Differently from KLAIM, muKLAIM allows dynamic privileges acquisition. In particular, this can be done by performing in/rd operations with special template fields. While in KLAIM security policies do not change dynamically and then static type checking allows to test if running processes have the necessary access permissions to perform their operations, muKLAIM proposes a type system that evolves according to dynamic variations of security policies. The approach exploits a combination of static and dynamic type checking. In order to improve the system performance, the type inference rules perform a static type checking to those operations for which it is possible to statically test the presence of the necessary access permission. Klava introduces encrypted messages into the fields of the tuples and the matching rule allows the evaluation of messages encrypted into fields; the encryption of messages ensures that they can be read only by the allowed clients.

Other two proposals introduce security mechanisms in Lime [MPR01] (Linda in a Mobile Environment), one at the implementation level [HR03] and the other one by extending the formal model [CVV]. The former is a secure implementation of Lime that provides a password-based access control mechanism at the level of tuples and tuple spaces. More precisely, the password-based system on tuple space and tuples permits the access only to the authorized agents, that is those that know the password. In particular, password-based access permissions on
tuples can be associated to the read and to the removal operations. In the case an agent is allowed to remove a tuple (i.e. it knows the password associated to the removal operations), it has also the permission of reading that tuple. Finally, the implementation also provides a support for guaranteeing a secure interaction among the agents that exploits cryptography. The latter proposes a formal model describing basic features of Lime and then extends it in order to support security. In particular, it is extended by adding a capability system that allows the owner of a tuple space to grant capabilities to that space. The proposed capability system has some similarities with KLAIM, indeed capabilities are at the level of tuple space and they express which operations are permitted to each process allowed to access the tuple space. This access control mechanism can be compared with the access permissions on locations described in the KLAIM model.

Finally, a different approach is proposed in [ORV03], where a general model for coordination middlewares that exploits process handlers to control the behaviour of processes has been presented. More precisely, a context is associated to each process and it defines which operations are allowed to the process. In order to express allowed actions, a language derived from CCS is used to describe which operations processes can perform.

1.5 Aims and Scope of the thesis

The main aim of the thesis is to investigate the security aspects outlined in the previous sections. In particular, we start by comparing the most interesting solutions presented in the literature and then we introduce SecSpaces, a novel proposal that aims at supporting some forms of control of the accesses to the data stored inside the tuple space.

SecSpaces follows the approach of the SecOS model; it associates permissions, that can be dynamically acquired, to each tuple and, instead of using field and object locks, decorates tuples with two additional control fields, namely the partition and the asymmetric partition fields. We consider this approach very
suitable for open systems because it is not required the users administration as in KLAIM; indeed, in this way any user can access the shared space without any registration or authentication phase that is needed in KLAIM where access permission are associated to each location. The main contribution of SecSpaces is a refinement of the access permissions on tuples that make it possible to discriminate between the permissions of producing, reading and consuming a tuple.

Chapter 2 contains a detailed comparison between the SecOS and the KLAIM approach, the description of the access control mechanisms of SecSpaces and the definition, in the tuple-space coordination model, of the main security properties (e.g., privacy, entity authentication) by showing that they can be guaranteed by implementing interaction protocol using the SecSpaces coordination primitives. Since these security issues are independent of the architecture of the Space Manager, we will not refer to a specific implementation of the space, then it may be based on a centralized solution (i.e. the space is stored in a single location), to be distributed on several locations or to be based on self reconfigurable systems, such as peer-to-peer systems [CFG+01].

We also define a complete framework comprising, besides the coordination model, tools for the formal reasoning about applications based on the SecSpaces model. In particular, in Chapter 3 the operational semantics of processes interacting via SecSpaces primitives and a notion of observational equivalence on systems, namely testing equivalence, are presented. We describe, by using some examples, how to exploit the observational equivalence in order to check some of the security properties outlined in the previous chapter.

The second step on security is to move our interest from the SecSpaces specification and theoretical aspects to those related with the implementation issues, taking into account the aspects described in Section 1.3. In Chapter 4 we propose an implementation of SecSpaces that has been applied to the emerging networking technology of the Web services. In particular, we discuss the implementation details of WSSecSpaces, a Web service supplying SecSpaces coordination primitives. In this context, we also give a new definition of entry, tem-
plate and consequently of the matching rule, that exploits XML-technologies to describe their structure and content, that allows for the definition of typed fields and typed wildcards. In appendix A we report the salient information about the offered Web services coordination service, that is: the definition of the structures used to represent entries/templates and the interaction modalities that should be used to invoke the SecSpaces primitives.

The access control mechanisms available in the literature, that we have discussed in the previous sections, do not permit to revoke access permissions and, usually, it is rather difficult to manage the restriction of the access permissions allowed to the processes. In Chapter 5 we present a simple extension of SecSpaces that supports a more sophisticated permission management. In particular, we describe how to manage group communication services that are usually supported by group-key management systems, such as the removal of some users from the group or the restriction of the communication to a subgroup of users. Notably, the proposed model presents some similarities with multilevel systems, in this chapter we define a general model of multilevel system based on one-way flow and then we show how it can be encoded by using the extended version of SecSpaces.

Linda tuple spaces are flat and unstructured, in the sense that they do not allow for expressing preferences of tuples; this feature can be undesiderable when many tuples match the template and some of them can be more important than other ones. Let us consider the classic example of master-worker applications in which masters submit job requests by producing tuples into the space and workers satisfy the requests by reading the tuples. Some of the submitted jobs could be more important than other ones or should be executed before other ones. More in general, it is not possible to express that some tuples should be returned more frequently w.r.t. other ones, or even tuples with a low relevance that should be taken under consideration only if there is no tuple with a higher importance. In Chapter 6 we investigate, in a process algebraic setting, how probabilities and priorities can be introduced in the Linda coordination model in order to support
a more sophisticated data retrieval mechanism. We present the semantics of the probabilistic data-retrieval primitives and of two forms of priority on tuples. The former assumes a static order relation among priority levels, while in the latter processes can express the order relation dynamically when the data-retrieval primitives are performed.

Finally, in Chapter 7 we investigate the possible combinations among the security mechanisms provided by SecSpaces, the probabilistic and the prioritized data-retrieval semantics. Indeed, priorities on tuples share interesting features with other Linda extensions supporting security properties. We describe how to combine the extended version of SecSpaces with priorities on tuples, depicting some possible scenarios in which it should be exploited. Another interesting case we analyse is the extension of SecSpaces obtained by adding probabilities on data-retrieval primitives. In order to prove that some applications can take advantage from this extension, it will be exploited to implement a registry for Web services supporting: i) run-time service discovery, ii) the control of the accesses to the Web services, and iii) a balanced workload among Web services supplying the same task. Since the SecSpaces model has already been implemented as a Web service, some comments about a prototypal implementation of the probabilistic version of WSSecSpaces are given.

1.6 Related publications

The contribute and the chapters of the thesis have already been published during the Ph.D. studies. The paper [BGLZ03] “SecSpaces: a Data-driven Coordination Model for Environments Open to Untrusted Agents”, accepted to the 1st International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA 2002), presents the idea of the access control mechanisms of SecSpaces and a comparison with related work. An extended version of the paper [GLZ], that is actually under consideration for a publication in the Fundamenta Informaticae journal, corresponds to Chapter 2.
Chapter 3 contains the work [BGL03] “A formal approach for checking security properties in SecSpaces”, accepted to the 1st International Workshop on Security Issues in Coordination Models, Languages and Systems (Secco’03).

The paper [LZ04] “WSecSpaces: a Secure Data-Driven Coordination Service for Web Services Applications”, accepted to the ACM Symposium on Applied Computing (SAC’04) at Coordination Models, Languages and Applications track, is reported in Chapter 4.

Chapter 5 is the result of a work that has been submitted for publication in the ENTCS volume of the MEFISTO project “Metodi Formali per la Sicurezza e il Tempo” (in english, formal methods for security and time).

Probabilities and priorities in Linda described in Chapter 6 are the results reported in [BGLZ04a] “Probabilistic and Prioritized Data Retrieval in the Linda Coordination Model”, accepted for publication to the 6th International Conference on Coordination Models and Languages (Coordination 04).

Finally, the paper [BGLZ04b] “Web Services for E-commerce: guaranteeing security access and quality of service”, accepted to the ACM Symposium on Applied Computing (SAC’04) at E-Commerce Technologies track, is contained in Chapter 7.
Chapter 2

Security issues in the tuple space coordination model for Open Environments

In this chapter we investigate security problems which occur when exploiting a Linda-like data driven coordination model in an open environment. In this scenario, there is no guarantee that all the agents accessing the shared tuple space are trusted. Starting from a formalization of some typical security properties in the standard Linda coordination model, we present a novel data-driven coordination model which provides mechanisms to support the considered security properties. The first of these mechanisms supports logical partitions of the shared repository: in this way we can restrict the access to tuples stored inside a partition, simply by limiting the access to the partition itself. The second mechanism consists of adding to the tuples some extra information which permits to authenticate the producer of a tuple or to identify its reader/consumer. Finally, we support the possibility to define access control policies based on the kind of operations an agent performs on a tuple, thus discriminating between (destructive) input and (non-destructive) read permissions on each single tuple.

2.1 Introduction

Starting from the considerations of Chapter 1, in this chapter we propose an extension/modification of the tuple space coordination model in order to sup-
port some form of control in the execution of the coordination primitives that the agents can perform to access the TS.

We have already discussed (see section 1.4.1, 1.4.2 and 1.4.3) the main proposals available in the literature, here we present a comparison between KLAIM and SecOS and then, by starting from these considerations, we present our proposal.

KLAIM exploits classic access control policies to manage the access of agents to tuple spaces: types are used to indicate the access rights of the agents, i.e., the operations that each agent 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 contents, but it can read all of the other tuples.

SecOS 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.

Comparing the approach of 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 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 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).

On the other hand, in SecOS it is not possible to express the access permission at the level of single primitive. Indeed, the read operator is defined as the juxtaposition of an input followed by an output operation, namely

\[ \text{rd } e \times P \overset{\text{def}}{=} \text{in } e \times (\text{out } x | P) \]

This fact has the two following consequences:

- There is no discrimination between non-destructive and destructive input access policies. More precisely, there is no way to discriminate between the readers and the consumers of a datum. This could be an undesired feature in many applications. Consider, e.g., a system in which an information can be accessed by any agent, but it can be removed only by specialized garbage collectors.

- An agent able to access a datum is also able to reproduce that datum, possibly creating new instances of that datum. This means that an agent which is a reader of a datum, is implicitly also a potential producer for that datum. Also this feature could be undesired in many applications. Consider, e.g., a master/worker system in which we want to ensure that tasks can be produced only by masters and consumed only by workers. According to the SecOS approach, there is no direct way to avoid a malicious worker to reproduce new instances of the tasks it consumes.

Starting from these observations, we propose a novel coordination model called SecSpaces which aims at supporting the advantages of both the approaches. In particular, as in SecOS we introduce the access information inside the data, but we refine the access controls supported in SecOS by permitting to discriminate, e.g., the \text{rd} from the \text{in} permissions.
We introduce SecSpaces incrementally. We initiate by reporting in Section 2.2 a formal representation of our starting point, that is the Linda coordination model. We first formally define the coordination medium, i.e. the TS, then we formally define the effect on the TS of the execution of the coordination primitives. Afterwards, we introduce a framework that permits to model some of the security properties typically considered for channel based communication. Namely, we consider data secrecy, data integrity, and data availability. We conclude our analysis of standard Linda by showing that the considered security properties (in particular data availability) cannot be ensured. Our proof permits to show that the basis of the security lacks of Linda are in the possibility to use wildcards in the pattern matching algorithm used in data retrieval operations. For example, each tuple in the TS can be read or even consumed by any of the agents in the system, simply by using wildcards accordingly.

In Section 2.2 we introduce a first extension of the standard Linda pattern matching. We consider an extra information, associated to the tuples, that cannot be matched by wildcards. Thus, in order to access a tuple, an agent must show a proof of knowledge of this extra information. We call this extra information partition field because its effect is to logically partition the TS. By means of a simple example, we show that data secrecy, data integrity and data availability now can be supported simply by exchanging the data through private partitions.

However, we show that partition fields are not sufficient for programming those applications in which it is necessary to authenticate the producer of a certain datum, as well as to identify the possible readers/consumers of that datum. For example, consider a client-server application in which the clients ask for services emitting tuples, and the servers execute the services described in the tuples. According to the access control supported by the partition fields, a malicious client could remove the service requests produced by any other client before that the server execute them. To solve this lack of the model, in Section 2.4 we introduce a more sophisticated access mechanism to the logical partitions, which is based on two asymmetric partition handlers: the first one to be used for data
introduction, and the second one for data retrieval. We call these handlers asymmetric partition fields. We show by means of a simple example, that this enhanced mechanism supports the authentication of the producers as well as the identification of the readers/consumers of the data exchanged through these new form of logical partitions.

Finally, in Section 2.5 we observe that the two mechanisms above do not permit to distinguish between the access policy to use for non-destructive and destructive access operations. For example, an agent able to read a datum is also able to remove that datum from the TS. Thus, we complete the definition of our new coordination model SecSpaces by associating to each tuple a pair of partition fields, and a pair of asymmetric partition fields. The first element of the pairs is considered when non-destructive read operations are executed, the second is used when destructive input operations are performed.

The main contribution of SecSpaces is in the definition of an advanced matching rule which, exploiting the three mechanism described above, is the basis of a novel approach for programming, in a more secure way, distributed applications based on tuple spaces. The examples that we report in the chapter prove the generality and the flexibility of this matching rule, that can be used to achieve several access control mechanisms. However, acting only at the level of the matching rule, we do not support any control on the execution of the output operations. Due to the asynchrony of the communication supported by the tuple space, we consider the problem of controlling the ability to produce tuples as a different and orthogonal task with respect to the analysis of advanced matching rules to be used to retrieve those tuples.

The last two Sections 2.6 and 2.7 report a more detailed comparison with related work and some conclusive remarks, respectively.
2.2 Security Properties in Linda

In this section we initiate our investigation of the security problems which arise when exploiting Linda-like tuple spaces for supporting the coordination of agents in an open application. We first introduce a formalization of the standard Linda coordination model, then we redefine in this new scenario some of the typical security properties which are usually considered in the context of secure channel communication.

2.2.1 Modeling the coordination medium

Linda supports coordination via a shared tuple space, where a tuple is a finite sequence of data.\(^1\) Formally, let Data, ranged over by \(d, d_1, \ldots\), be a denumerable set of data. Let Entry, ranged over by \(e, e', \ldots\), be the set of finite sequences of data taken from Data and denoted by \(\langle d_1, \ldots, d_n \rangle\), with \(n \geq 1\).

In order to perform input and read operations, templates are used by an agent to specify the kind of entry the agent is interested in. Templates may use wildcards to let some field unspecified; syntactically, a wildcard is denoted by \texttt{null} (which we assume to be not in Data). In the following, \(dt, dt_1, \ldots\), range over Data \(\cup \{\texttt{null}\}\). Formally, let Template, ranged over by \(t, t', \ldots\), be the set of finite sequences composed of data and wildcards denoted by \(\langle dt_1, \ldots, dt_n \rangle\), with \(n \geq 1\).

A matching rule between entries and templates is used to indicate when an entry satisfies a certain template. The standard Linda matching rule is as follows.

**Definition 2.1 Matching rule** - Let \(e = \langle d_1, \ldots, d_n \rangle\) be an entry and \(t = \langle dt_1, \ldots, dt_n \rangle\) be a template; we say that \(e\) matches \(t\) if the following condition holds:

- for each \(1 \leq i \leq n\), either \(dt_i = d_i\) or \(dt_i = \texttt{null}\).

\(^1\) In real Linda implementations, a tuple is a finite sequence of typed fields; for the sake of simplicity we consider a unique generic type Data.
The coordination primitives we consider are the classical Linda operations: \texttt{out}(e) which introduces the entry \(e\) in the shared repository; \texttt{rd}(t) which returns a copy of a matching entry \(e\) (i.e. \(e\) matches \(t\)) which is currently available in the shared repository; \texttt{in}(t) which is the destructive counterpart of \texttt{rd}(t). For the sake of simplicity, we usually omit the angular parenthesis from the denotation of \texttt{in} and \texttt{rd} operations: e.g. we use \(\texttt{in}(d_1, \text{null})\) instead of \(\texttt{in}((d_1, \text{null}))\).

Observe that the operations, excluding \texttt{out}(e), have a return value: \texttt{rd}(t) and \texttt{in}(t) return a copy of the matching entry \(e\) (in the case of \texttt{in}(t) this matching entry \(e\) is also removed from the repository).

Let \(\text{Action} = \{\text{out}(e), \text{rd}(t), \text{in}(t) \mid e \in \text{Entry}, t \in \text{Template}\}\) (ranged over by \(\alpha\)) be the set of operations that an agent can perform, and let \(\text{ReturnValue} = \text{Entry} \cup \{-\}\) (ranged over by \(r\)) be the set of return values of the operations. We use \(-\) to denote the empty return value of the \texttt{out} operation.

The current state of the shared tuple space is modeled by a multiset of entries: formally, the state of the tuple space is modeled with \(\mathcal{M}(\text{Entry})\), i.e. the set of the possible multisets on \(\text{Entry}\). To describe the effect of the operations on the shared repository, we use a labeled transition system \((\mathcal{M}(\text{Entry}), \text{Label}, \rightarrow)\) where \(\text{Label} = \text{Action} \times \text{ReturnValue}\). The typical transition of this labeled transition system is denoted with \(\text{TS} \xrightarrow{\alpha} \text{TS}'\), whose meaning is as follows: when the state of the tuple space is \(\text{TS}\), and an action \(\alpha\) is performed, then the new state of the repository is \(\text{TS}'\) and the return value is \(r\). The labeled transition system is defined as the smallest one satisfying the rules in Table 2.1.

The first rule describes the effect of the execution of an \texttt{in}(t) operation: one of the available matching entry \(e\) is removed from the \(\text{TS}\), and it is returned as result. The second rule describes \texttt{rd}(t) operation: observe that the returned entry is not removed from the tuple space. The last rule describes the execution of an \texttt{out}(e) operation whose effect is to add the entry \(e\) to the tuple space.

It is interesting to observe that wildcards can be used to access all the information available in the tuple space. Consider, e.g., a generic entry \(\langle d_1, d_2 \rangle\) that
can be read by any agent simply performing \( \text{rd}(d_1, \text{null}) \) or even \( \text{rd}(<\text{null}, \text{null}>) \). In particular, the operation \( \text{rd}(<\text{null}, \text{null}>) \) exploits wildcards to match with all possible entries having exactly two data fields.

This simple example shows the main difficulty that the application developer encounters when he needs to bound the scope of visibility or to reduce the accessibility to information which are introduced in an entry of the tuple space. The typical solution to this problem could be to encrypt the information, in such a way that the information can be decoded only by the expected receiver of the information. However, this is not completely satisfactory, because there is no guarantee that the information will be removed from the tuple space only by the expected consumers, because any agent could exploit wildcards to remove the entry containing the encrypted information. In other words, the model cannot ensure that the entry is always available and delivered to the expected agent. This problem will be discussed and formalized more in detail in Section 2.2.3. In the remainder of this section we formalize our investigation of these security problems encountered when the standard Linda coordination model is adopted.
2.2.2 Modeling the agents

In order to introduce a formal definition of the security properties we need to provide, besides the modeling of the tuple space reported above, also a modeling of the agents in the system.

An agent has its own identity, an internal state, and a knowledge. The knowledge is used to formalize the information an agent is aware of, in order to be able to verify, e.g., whether an information is secret or known by all the agents in the system.

Formally, let \( A \), ranged over by \( A_1, A_2, \ldots \), be a denumerable set of agent identifiers; let \( S \), ranged over by \( S, S', \ldots \), be a denumerable set of states. The knowledge of an agent is usually denoted with the set of data it is aware of; formally, we use \( \phi, \phi', \ldots, \) ranging over \( \text{Knowledge} = \mathcal{P}(\text{Data}), \) i.e. the powerset of \( \text{Data} \). An agent with identity \( A \), a state \( S \), and a knowledge \( \phi \) is denoted with \( A_{\phi}^S \). A configuration of agents \( A_{\phi}, A_{\phi}', \ldots \), is a set of agents such that there exist no two different agents with the same identity; formally, \( A_{\phi}, A_{\phi}', \ldots \), range over the set

\[
\text{Agent} = \{ [A_{\phi_1}^S, \ldots, A_{\phi_m}^S] | \text{for each } i \neq j \text{ then } A_i \neq A_j \}
\]

In the following we will omit parenthesis for set of agents containing a single agent, e.g. we use \( A_{\phi}^S \) instead of \( \{A_{\phi}^S\} \).

In the following we will adopt the following notation:

- \( \text{arg}(\alpha) \) denotes, given an action \( \alpha \), the argument of \( \alpha \); that is, \( \text{arg}(\text{out} \{e\}) = e \) and \( \text{arg}(\text{op} \{t\}) = t \) if \( \text{op} \in \{\text{rd, in}\} \);

- \( \text{dataset}(e) \) and \( \text{dataset}(t) \) denote, given either an entry \( e \) or a template \( t \), the set of data fields stored inside it, e.g., \( \text{dataset}(\langle d_1, d_2 \rangle) = \{d_1, d_2\} \);

- \( D(\phi) \) denotes, given a knowledge \( \phi \), the entries and the templates that an agent with knowledge \( \phi \) can generate, more precisely, the entries and templates which contain the data in \( \phi \) or the wildcard \( \text{null} \); for example, \( \langle d_1, d_2 \rangle \) and \( \langle d_1, \text{null} \rangle \) are in \( D([d_1, d_2]) \) while \( \langle \text{null}, d_3 \rangle \) is not in \( D([d_1, d_2]) \).
In order to be as general as possible we do not specify the internal behaviour of the agents. We simply assume given a relation $\text{act}$ which indicates the possible actions that an agent can take according to its current state and knowledge.

Formally, let $\text{act} \subseteq \text{State} \times \text{Knowledge} \times \text{Action} \times \text{ReturnValue} \times \text{State}$ be a relation composed of 5-ples $(S, \phi, \alpha, r, S')$ having the following meaning: if an agent has an internal state $S$ and a knowledge $\phi$, then the agent is willing to perform the action $\alpha$ and, in the case the return value is $r$, then the next state will be $S'$. Note that $\text{act}$ does not depend on the agent identifier, hence agents having the same state (and the same knowledge) are behaviourally equivalent.

We make the following two assumptions on any of the 5-ples $(S, \phi, \alpha, r, S')$ of $\text{act}$. The former is that the argument of the action $\alpha$ should be compliant with the current knowledge $\phi$ of the agent; formally, we assume that $\text{arg}(\alpha) \in \text{D}(\phi)$. The latter is that the next state should be defined for any possible return value $r'$, even if different from $r$; formally, if $\text{TS} \xrightarrow{\alpha} \text{TS}'$ (for some $\text{TS}$ and $\text{TS}'$), then also a 5-ple $(S, \phi, \alpha, r', S'')$ is in $\text{act}$ for any $r' \neq r$.

We are now ready to formalize a system as a pair $(\text{Ag}, \text{TS})$ composed of a set of agents $\text{Ag}$ and the current state $\text{TS}$ of the tuple space; formally, let $\text{System} = \text{Agent} \times \mathcal{M}(\text{Data})$. The semantics of the systems is described by a labeled transition system $(\text{System}, \text{Label}, \rightarrow)$, where $\text{Label} = \text{Action} \times \text{ReturnValue} \times \text{AgentId}$, with transitions denoted as $(\text{Ag}, \text{TS}) \xrightarrow{\alpha} \text{Ag}'$, $(\text{Ag}, \text{TS}) \xrightarrow{\alpha} \text{Ag}'$, and $(\text{Ag}', \text{TS}')$. The interpretation for the transition is that given a system $(\text{Ag}, \text{TS})$, then this can evolve to $(\text{Ag}', \text{TS}')$ as effect of the execution of the operation $\alpha$, with return value $r$, performed by the agent with identity $A$.

The labeled transition system is defined as the smallest one satisfying the rule in Table 2.2.

It is worth noting that the knowledge of the agent performing the operation may change in the case new information is achieved as result of a read (or input) operation.

With an abuse of notation, we use $\rightarrow$ for both the transition systems defined.
Table 2.2: Semantics of the agents

in Tables 2.1 and 2.2. However, this does not introduce confusion because the current meaning of \( \rightarrow \) will be made clear from the context.

In the following we will use \( (A_g, TS) \rightarrow^{*} (A_g', TS') \) to denote the fact that the system \( (A_g, TS) \) may evolve to \( (A_g', TS') \); namely, there exists a sequence of transitions

\[
\begin{align*}
(A_g, TS) &\rightarrow_{A_{\alpha_1}} (A_{g_1}, TS_1) \\
&\rightarrow_{A_{\alpha_2}} ... \\
&\rightarrow_{A_{\alpha_{n-1}}} (A_{g_{n-1}}, TS_{n-1}) \\
&\rightarrow_{A_{\alpha_n}} (A_{g'}, TS').
\end{align*}
\]

We write \( (A_g, TS) \rightarrow_{A} \) to state that there exists \( (A_g', TS') \) such that \( (A_g, TS) \rightarrow_{A} (A_{g'}, TS') \).

2.2.3 Formalizing the security properties

We have already observed that when tuple spaces are used as coordination infrastructures in open applications, security problems occur due to the fact that also untrusted agents may access the tuple space, and perform any kind of coordination primitives. In order to formalize the effect of the presence of these untrusted entities in the system, we consider a specific agent, called the \textit{enemy}, having the ability to execute any possible action at any possible instant. The enemy is formally modeled as an agent with a specific identity \( E \) and a specific state \( S_E \). In order to permit to the enemy to perform all the possible actions that
the untrusted unexpected entities may execute, we assume that the state $S_E$ has the following property: for any knowledge $\phi$ and any return value $r$, the set $\{(S_E, \phi, \alpha, r, S_E) \mid \text{arg} (\alpha) \in D (\phi)\}$ is a subset of $\text{act}$, i.e. this state allows to nondeterministically perform all the operations permitted by the current knowledge of the agent.

We are now ready to formally define, in our scenario, three security properties. Namely, we consider data secrecy, data integrity (also called data authenticity, see [FGM00]) and data availability. Informally, secrecy corresponds to the ability of a secure channel to maintain confidential, i.e. unknown to the enemy, the information which are exchanged through the channel. Data integrity means that the contents of the data which are communicated through the channel cannot be modified by any possible enemy. Finally, data availability means that the enemy cannot remove any datum from the secure channel.

In order to formalize data secrecy, we consider a specific system with a confidential datum; then, we considered the same system extended with the enemy, and we require that the enemy has no way to add the confidential datum to its knowledge.

**Definition 2.2  Data secrecy** - Let $(A g, TS)$ be a system, and let $\phi_E$ be the initial knowledge assigned to the enemy $E$. Let $d \in \text{Data}$ be a datum: we say that Secrecy$(\langle A g, TS \rangle, \phi_E, d)$ holds if and only if for any $\phi'_E$ such that $(A g' \cup E^{\phi'_E}_{S_E}, TS) \rightarrow^*(A g' \cup E^{\phi'_E}_{S_E}, TS')$, we have that $d \not\in \phi'_E$.

Observe that the test is performed assuming a specific datum $d$, whilst other secrecy definitions, see e.g. [AG99], extend the test applying the universal quantifier to $d$, requiring that the test holds for any $d \in \text{Data}$. However, our approach is more flexible because, when it is necessary, permits to extend the secrecy test by universally quantifying the datum $d$ on a specific subset of $\text{Data}$ (and not only on the whole set $\text{Data}$). For the same reason, also the following properties that we are going to define do not consider universal quantification on the data.
Note that data secrecy does not ensure that the data will always be delivered to one of the honest agents. Moreover, the knowledge of the agents in $\mathcal{A}_g$ is completely disregarded, that is, it is not important which agent knows the secret datum. This reflects the fact that, in this context, the same datum (even if confidential) can be non-destructively accessed by more than one agent. On the other hand, in secure channel communication we usually assume that each channel has a single expected reader and a single expected consumer.

We now consider data integrity. Suppose that in a system $(\mathcal{A}_g, \mathcal{S})$ there is a datum $M$ that an agent $A$ should read. The integrity of the datum is supported only if it is ensured that the agent $A$ receives exactly $M$, and not a different datum $M'$. In order to formally check that data integrity is supported, following the approach used in [FGM03], we assume that after the agent $A$ has read the datum for which integrity is assumed, it communicates this datum, say $d$, by performing the operation $\text{out}(\langle \text{commit}, d \rangle)$. Integrity is actually guaranteed if it is ensured that $d$ corresponds to the expected $M$.

**Definition 2.3** Data integrity - Let $(\mathcal{A}_g, \mathcal{S})$ be a system, and let $\phi_E$ be the initial knowledge assigned to the enemy $E$. Let $M \in \text{Data}$ be a datum. Integrity$(\langle \mathcal{A}_g, \mathcal{S} \rangle, \phi_E, M)$ holds if and only if for any agent identifier $A \neq E$ such that $(\mathcal{A}_g \cup E^\phi_E, \mathcal{S}) \xrightarrow{*} \text{out}(\langle \text{commit}, d \rangle) \to_A (\mathcal{A}_g' \cup E^\phi_E, \mathcal{S}')$ then $d = M$.

In line with the observation reported above, several agents could read the datum $M$; this is formally reflected by the fact that the agent identity $A$ is universally quantified in the formal definition. It is worth noting that the formal definition does not consider the fact that $M$ is not in the knowledge of the enemy, hence data integrity does not ensure the confidentiality of datum $M$.

The last security property that we consider is data availability. This property is important when we would like to ensure that a datum $d$ cannot be removed by the enemy. Suppose that $\text{Ids} = \{A_1, \ldots, A_n\}$ is the set of the identities of the agents interested in reading the datum $d$. Data availability is ensured if, after that
Chapter 2. Security in tuple space coordination model

the datum is introduced in the tuple space, it remains available at least until all the agents have accessed the datum.

**Definition 2.4 Data availability** - Let \((A_{g_0}, TS_0)\) be a system, and let \(\phi_E\) be the initial knowledge assigned to the enemy \(E\). Let \(\text{Ids}\) be a set of agent identities and let \(d \in \text{Data}\) be a datum.

Availability \((\langle A_{g_0}, TS_0 \rangle, \phi_E, \text{Ids}, d)\) holds if and only if for any agent identifier \(A \in \text{Ids}\) and any possible computation of the system with enemy

\[
(A_{g_0} \cup E_{\phi_E}^0, TS_0) \xrightarrow{\alpha_1} (A_{g_1} \cup E_{\phi_E}^{\phi_{\text{E}_0}}, TS_1) \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} (A_{g_n} \cup E_{\phi_{\text{E}_n}}, TS_n)
\]

one of the three following conditions holds:

- the datum \(d\) is not introduced in the tuple space, i.e., \(d \notin \bigcup_{e \in TS_t} \text{dataset}(e)\) for any \(0 \leq i \leq n\);

- after that the datum \(d\) is introduced in the tuple space, the agent \(A\) has always the capability to read/consume it; formally, if \(0 \leq i \leq n\) such that \(d \in \bigcup_{e \in TS_t} \text{dataset}(e)\), then for any \(j \geq i\) we have that there exist \(e \in TS_j\) (with \(d \in \text{dataset}(e)\)) and a template \(t \in D(\phi_{A_{t_i}})\) (where \(\phi_{A_{t_i}}\) is the knowledge of the agent \(A\) in \(A_{g_j}\)) such that \(e\) matches \(t\);

- the agent \(A\) accesses the datum \(d\) during the step \(j\) of the computation; formally, there exists \(1 \leq j \leq n\) such that \(A_{t_i} = A\) and \(d \in \text{dataset}(r_i)\).

Also in this case the formal definition does not assume that the datum \(d\) is not in the knowledge of the enemy; thus, data availability may be ensured even if the enemy can access the datum \(d\). Moreover, the definition does not make any assumption on the number of occurrences of the datum \(d\); hence, data availability may be ensured even if the enemy removes datum \(d\), provided that it has no way to remove all of its instances.

We are now ready to formalize the main general security lacks of the Linda coordination model. We will consider data secrecy and availability, while we do
not take data integrity into account. Indeed, integrity is a property which strongly
depends on the specific communication protocol executed by the agents, and it
is not a general property. For example, a system in which no agent performs the
out(⟨commit, d⟩) operation trivially satisfies data integrity.

**Proposition 2.1** Let \((A, TS)\) be a system and \(I\) a set of identities of agents in \(A\). For
any \(d \in \bigcup_{e \in TS} \text{dataset}(e)\) we have that Secrecy\((A, TS), \phi, d\) and Availability\((A, TS), \phi, I, d)\) do not hold.

**Proof:** As \(d \in \bigcup_{e \in TS} \text{dataset}(e)\), we have that there exists an entry \(e\) in \(TS\), such
that \(e = \langle d_1, \ldots, d_n \rangle \in TS\) and \(d = d_i\) for some \(1 \leq i \leq n\). Consider now the
template \(t = \langle \text{null}, \ldots, \text{null} \rangle\) with exactly \(n\) wildcards. Trivially we have that \(t \in D(\phi)\), thus the following computation can occur
\(\langle A \cup E_{\phi_E}^{\text{null}} \cup \text{null}^n, TS \rangle \xrightarrow{\text{null}}_E \langle A \cup \text{null}^n \cup \text{null}^n, TS \rangle\). We have that \(d \in \phi_E\), thus secrecy is not supported.

The same reasoning can be used in order to prove that also availability is not
supported. It is enough to consider all entries \(e_1, \ldots, e_m\) containing the datum \(d\),
and observe that the enemy can use wildcards and the \text{in} operation in order to
remove all these entries.

This proposition shows that wildcards, and the matching rule, are the basis for
the security lacks of Linda. A first approach for resolving these problems could
be to exploit standard cryptographic techniques [Sch96] in order to introduce en-
crypted data inside tuples. This approach could, e.g., support secrecy, provided
that the enemy has no ability to decrypt the confidential data. However, the so-
lution is not completely satisfactory because availability cannot be ensured.

In the remainder of the chapter we present extensions of the matching rule,
and we will show that these extensions will permit to support secrecy and avail-
ability even without exploiting cryptographic techniques.
2.3 Partitioning the Tuple Space

As observed in the previous section, in Linda systems each tuple in the shared repository may be potentially read or consumed by each agent having access to that repository. In many applications this could be an undesired feature; consider, e.g., applications in which the agents are divided into classes and it is necessary to distinguish among the entries of one class and the entries of another class. A possible approach for supporting this separation could be to use one repository for each group; however, this could lead to an undesired proliferation of repositories, one for each group to be supported. In this section we present a simpler approach which permits to logically divide the whole dataspace in separated partitions.

2.3.1 Partition fields

The idea is to extend the entry structure with a special field, named *partition field*. The partition field is the only one in which the matching rule does not accept wildcards. Therefore, an agent executing a read operation on an entry with a partition field is forced to deliver a proof of knowledge of that partition field.

Formally, let Partition be a denumerable set of partition field values (ranged over by \( c, c_t, c_s, \ldots \)). We also assume that Partition contains a special default value \( \# \) used when the partition field has no importance (the actual use of \( \# \) will be described in the following). An entry \( e \), and a template \( t \) with the partition field, are defined as follows:

\[
e = \langle d_1, \ldots, d_n \rangle^{[c_t]}
\]

\[
t = \langle dt_1, \ldots, dt_n \rangle^{[c_t]}
\]

with \( n \geq 1 \). Let Entry\(^c\) and Template\(^c\) be the set of entries and templates with partition field, respectively. In the following, \( e, e', \ldots \), range over Entry\(^c\), and \( t, t', \ldots \), range over Template\(^c\). The tuples \( \langle d_1, \ldots, d_n \rangle \) and \( \langle dt_1, \ldots, dt_n \rangle \), \( n \geq 1 \), are also denoted in the compact form \( \langle \vec{d} \rangle \) and \( \langle \vec{dt} \rangle \), respectively. We also define \( e_{[d]} \) as the operator that, given an entry \( e = \langle \vec{d} \rangle^{[c_t]} \), returns the tuple of data \( \langle \vec{d} \rangle \).
Definition 2.5 Matching rule with partition fields - Let \( e = \langle \tilde{d} \rangle^{|c|} \) be an entry, and \( t = \langle \tilde{d}t \rangle^{|c'|} \) be a template; we say that \( e \) matches \( c^e \) \( t \) when the following conditions hold:

- \( c_t = c \),
- \( \langle \tilde{d} \rangle \) matches \( \langle \tilde{d}t \rangle \).

Observe that the definition of \( \text{matches}^e \) is a conservative extension of \( \text{matches}^c \) (see Definition 2.1).

A consideration about performance is of interest. Given a template \( \langle \tilde{d}t \rangle^{|c'|} \), if there exists a matching entry \( e \) in the space, then \( e = \langle \tilde{d} \rangle^{|c|} \) and \( c_t = c \). We can exploit \( c_t \) in order to reduce the search space to the partition identified by \( c_t \).

Example 2.1 \( \text{matches}^c \) evaluation - A list of entry matching examples follows:

| template                      | \text{does it match with } \langle d_1, d_2 \rangle^{|c|}? |
|-------------------------------|----------------------------------------------------------|
| \langle \text{null} \rangle^{|c|} | no                                                       |
| \langle \text{null, null} \rangle^{|c|} | yes                                                      |
| \langle d_1, \text{null} \rangle^{|c|} | yes                                                      |
| \langle d_1, d_2 \rangle^{|c|} | yes                                                      |
| \langle d_1, d_2 \rangle^{|c'|}, c' \neq c | no                                                       |
| \langle \text{null, null, null} \rangle^{|c'|} | no                                                       |

The new operational semantics, which takes into account the presence of the partition fields, is obtained by replacing in Table 2.1 \( \text{matches}^c \) with \( \text{matches}^e \), and the \( e \) used to describe the return value with \( e \mid d \) (in this context \( \text{ReturnValue} = \{ e \mid d \mid e \in \text{Entry}^e \} \cup \{ - \} \)). Observe that the return value does not contain the partition field, as we use this information only to control the access to the entries.

The partition field and the new matching rule induce a partitioning of TS. Every \( c \in C \) identifies a subspace, which is composed of all those entries in TS with partition field set to the value \( c \). Clearly, in order to access the subspace characterized by \( c \), an agent must know \( c \). However, the special value \# provides
the agents with a default subspace to be used when no specific partition has been defined as working space. For the sake of simplicity, when an entry (template) has the partition field set to #, we omit it. For example, the Linda system defined in the previous section can be seen as a subcase of the new system, provided that only the default partition # is used.

In order to adapt to the new setting the security properties defined in the previous section, we need to modify the definition of the agents. First of all, we need to redefine the representation of knowledge, taking also partition fields into account: Let $\text{Knowledge} = \mathcal{P}(\text{Data}) \cup \mathcal{P}(\text{Partition})$. Under the new interpretation of knowledge, $D(\phi)$ returns the set of entries and templates that can be composed using as data fields the data in $\phi$ (or `null`) and as partition field one of those in $\phi$ (or #). For example, $\langle d_1, \text{null} \rangle^{c} \in D([d_1, c, c'])$ and $\langle d_1, d_2 \rangle^{c,c'} \not\in D([d_1, d_2, c, c'])$. The functions $\text{dataset(e)}$ and $\text{dataset(t)}$ returns the set of data fields stored inside the argument, hence it does not return the partition field. Once adopted these new definitions, the semantics of the new systems is given by the smallest labelled transition system satisfying the rule in Table 2.2.

We have previously observed that the partition field induces a partitioning of $TS$. This fact can be used in order to limit the access to a partition; it is enough to distribute the knowledge of the corresponding partition field value to the group of agents that has access permission to the partition. Exploiting this form of access control, the following example shows that, differently from Linda, we can define a system composed of two agents, say $A$ and $B$, in which $A$ sends an entry to $B$ and the data are exchanged in a secure way.

**Example 2.2 Private partition** - Let $(Ag, TS)$ be a system with empty tuple space, i.e. $TS = \emptyset$, and two agents $A$ and $B$ which exchange a datum through a partition with name $c_s$. More precisely, $Ag = \{A_{S_A}^{d,c_s}, B_{S_B}^{commit,c_s}\}$. Suppose that the first agent produces the entry $\langle d \rangle^{c_s}$, while the second one consumes an entry performing an $in(\langle \text{null} \rangle^{c_s}, r, S_B)$ operation, and then communicates the consumed datum, say $d'$, producing the entry $\langle \text{commit}, d' \rangle^{c_s}$. Formally, we assume that the unique actions the agents can perform are
(S_A, [d, c_s], out ([d]^{c_s}), -, S_t) \in act

(S_B, [commit, c_s], in ([null]^{c_s}, r, S'_B) \in act \text{ (for any } r)

(S'_B, [commit, d', c_s], out ([commit, d']^{c_s}), -, S_t) \in act \text{ (for any } d')

where we assume that S_t is a terminating state, i.e. a state that does not permit the execution of any action.

If we assume that only A and B know c_s, we have that the partition corresponding to c_s can be accessed only by the two agents. In other words, if \phi_E is the initial knowledge of the enemy, and if c_s \not\in \phi_E, we have that all the properties Secrecy\left([A_g, TS], \phi_E \setminus [d], d\right)^2, Integrity\left([A_g, TS], \phi_E, d\right) and Availability\left([A_g, TS], \phi_E, [B], d\right) hold.

The proofs are rather simple, we just remind that, according to the definition of matches\textsuperscript{c}, in order to read or remove the entry written by A the enemy must be able to generate a template having the partition field set to c_s. But this is not possible because c_s \not\in \phi'_E for each \phi'_E such that \left(A_g \cup E^{\phi_E}_{S_E}, TS\right) \rightarrow^* \left(A_g' \cup E^{\phi'_E}_{S_E}, TS'\right). Consequently, secrecy and availability hold because the enemy cannot neither read nor remove that entry.

To prove data integrity, we reason in a similar way, observing that the enemy cannot introduce any entry with partition field set to c_s. Thus, for each TS' such that \left(A_g \cup E^{\phi_E}_{S_E}, TS\right) \rightarrow^* \left(A_g' \cup E^{\phi'_E}_{S_E}, TS'\right), we have that the unique entry with partition field set to c_s is [d]^{c_s}. This ensures that the unique datum that the agent B may commit is the expected datum d.

Finally, note that in order to prove the properties we have not made any assumption on d, hence they hold for any d \in Data.

Following this approach, we may implement also a form of secure group communication, simply by limiting the knowledge of a specific partition to the group of agents that need to communicate in a secure way.

More precisely, suppose we have the groups of agents \text{G}_1, \ldots, \text{G}_n. We associate to each group \text{G}_i a partition name c_i \in C, with c_i \neq c_j for any i \neq j. We

\textsuperscript{2}Obviously, if d is in the initial knowledge of E Secrecy does not hold.
assume that $c_i$ is known only by the agents that are members of $G_i$, and that they keep $c_i$ secret.

In order to write or consume a datum into the space dedicated to $G_i$, it is sufficient to execute $\text{out}(\langle d \rangle^{c_i})$ or $\text{in}(\langle d \rangle^{c_i})$, respectively. The \texttt{rd} operation can be treated similarly to \texttt{in}. Following the proofs above, it is possible to show that, if the knowledge of the enemy does not contain any of the partition fields used by the groups, the data exchanged in those partitions satisfy secrecy, integrity and availability.

### 2.3.2 Scope of the partition fields

In the Example 2.2 we have assumed that the initial tuple space $TS$ is empty. This assumption is rather unnatural, because in general applications running in open systems cannot impose the state of the tuple space when they start. However, the assumption is important for the proofs of the security properties because, in the case the partition name $c_s$ is stored in some of the entries in the tuple space, the enemy could add it in its knowledge. Another important assumption we make in the example is that both $A$ and $B$ keep the name $c_s$ secret, i.e. they do not introduce it in the tuple space.

We have made no assumption concerning the relation between $Data$ and $Partition$ so far, and this clearly influences the scope of partition fields. More precisely, consider that the two sets are separated, i.e. $\text{Partition} \cap \text{Data}$ is empty; in this case it is not possible to introduce in the tuples any partition field value. Under this assumption, we have that the scope of partition fields is static, i.e. the agents cannot change their knowledge of partition fields during the computation.

Under static scope of the partition fields, the assumptions taken in the example are not necessary because the enemy has no way to add to its knowledge the secret partition field $c_s$.

On the other hand, one could consider the possibility to introduce partition field values inside the tuples. Formally, we could consider $\text{Partition} \subseteq \text{Data}$. In
this case we have that the scope of partition fields is dynamic, because the agents can increase the amount of partition fields that they know.

Differently from the static case, under dynamic scope of the partition fields, the assumptions taken in the example are necessary; consider, e.g., the case in which TS contains an entry \( e \) such that \( c_s \in \text{dataset}(e) \). Using wildcards the enemy E can get the name \( c_s \), and then it can add, read or remove entries from the partition related to \( c_s \), thus invalidating the properties claimed in the example.

On the one hand, the static case offers the possibility to robustly control the access to the entries located in the partitions; on the other hand, the mechanism offers a limited flexibility because the permissions must be settled at the start-up of the system. On the contrary, the dynamic case is more vulnerable, but it permits a more flexible way to manage the access permissions; indeed, differently from the static case, they can be changed (even if only monotonically) every time the system needs to do it.

These two approaches clearly influence the way partition fields are distributed. E.g., in the Example 2.2 we have assumed the distribution of \( c_s \) as a precondition. In the static case, we have no manner to distribute the partition fields using the tuple space, thus another form of communication is needed to establish the partition to use. In the dynamic case, partition fields can be exchanged through the tuple space; the open problem is how to perform this phase in a secure way.

More precisely, when the partition fields are exchanged, it is important to authenticate the producer and the receiver of the entries containing this information. For example, an agent \( A \) that wants to communicate with \( B \) in a secure way through a fresh partition, i.e. a partition which is not currently used and known by the other agents in the system, settles this new partition and then communicates the corresponding partition field value through the tuple space. The agent \( A \) needs some guarantees that only \( B \) can read/consume this value from the tuple space, and also \( B \) wants to be sure to receive the name established by \( A \), and not other partition value produced by malicious agents.

In the following subsection we formally model the authentication property
of producer/receiver of an entry, and we prove that the partition fields are not expressive enough to ensure also these security properties.

Besides the distribution of the private partition field identifiers, another issue regards the “freshness” of the partition field. In Example 2.4 we will discuss a possible implementation of a partition field generator service that guarantees freshness and secure distribution of the partition values.

### 2.3.3 Producer and Receiver Authentication

In the Example 2.2 we have shown that private partitions can be used to exchange data in such a way that secrecy, integrity, and availability are supported. There exist other interesting properties that secure channels should usually support, e.g. the possibility to authenticate the producer and/or the receiver of a particular information communicated through the channel.

Consider, for example, an entry in the tuple space that represents an electronic payment sent from the agent \( /BT \) to the agent \( /BU \). The agent \( /BT \) wants to be sure that the entry will be received only by \( /BU \) and, on the other hand, the agent \( /BU \) requires reasonable guarantees that \( /BT \) is the actual producer of the entry. For example, it is important that \( /BT \) is not going to repudiate the payment, as well as that \( /BU \) is unable to recycle the payment, reintroducing it in the tuple space after its initial withdrawal. We refer to these security properties as producer authentication and receiver authentication, respectively.

Formally, we can model producer and receiver authentication, as follows.

**Definition 2.6 Producer and Receiver Authentication** - Let \((A_g, TS)\) be a system and consider two agents \( A_g^{\phi_A}, B_g^{\phi_B} \in A_g \). We assume \( A_g = \{A^{\phi_1}_1, \ldots, A^{\phi_n}_{n_S} \} \).

- Let \( A_i = A \) and \( e \) be an entry we want to test if only \( A \) can produce; we say that \( \text{ProdAuth}((A_g, TS), A, e) \) holds if and only for any computation

\[
([A^{\phi_1}_{S_E}, \ldots, A^{\phi_{i-1}}_{S_E}, A^{\phi_A}_{S_A}, A^{\phi_{i+1}}_{S_E}, \ldots, A^{\phi_n}_{S_E}], TS) \rightarrow^* (A_g', TS') \longrightarrow_C \text{ out}[e] \]

we have that \( C = A \);
Let $A_j = B$ and $e$ be an entry we want to test if only $B$ can read/remove; we say that $\text{RecvAuth}((A_g, TS), B, e)$ holds if and only if for any computation

$$(A^j_{i_S}, \ldots, A^j_{i_S}, A^j_{i_S}, A^j_{i_S}, \ldots, A^j_{i_S}, TS) \xrightarrow{\text{op}(t)} \tau_c$$

with $\text{op} \in \{\text{rd, in}\}$ and $e$ matches $^c t$, we have that $C = B$.

The first part of this property ensures that if an agent writes $e$, then it is $A$. Note that this condition is tested for any behaviour that the agents of the system can have (keeping their original knowledge). This is obtained by substituting the states $S_i$ with $S_F$. In this way we guarantee that, even if some components of the system are malicious, $A$ is the only agent that can write $e$. The second property regards the ability to read a specific entry $e$; it ensures that, even assuming malicious behaviour of the the agents different from $B$, if an agent performs a rd/in operation using a template that matches with $e$, then it is $B$.

It is interesting to remark that, differently from the security properties given in Section 2.2.3, we define the producer/receiver authentication without explicitly introducing any enemy agent, but we introduce malicious behaviours of the agents composing the system. The same property could be equivalently defined performing the test on the system with enemy simply by fixing its initial knowledge equal to the union of the single knowledges of the agents in $A_g$ (excluding that of $A$ for producer authentication, and that of $B$ for receiver authentication).

We are now ready to prove that the partition fields are adequate to support authentication only in the trivial case in which the producer and the consumer are the same agent.

**Proposition 2.2** Let $(A_g, TS)$ be a system. Let $A^j_{i_S}$ and $B^j_{i_S}$ be two agents in $A_g$ that want to exchange data via an entry $e = [\bar{d}]^{[c]}$ in such a way that both $\text{ProdAuth}((A_g, TS), A, e)$ and $\text{RecvAuth}((A_g, TS), B, e)$ hold. If there exists a communication session in which $A$ and $B$ exchange the entry $e$, i.e.

$$(A_g, TS) \xrightarrow{\text{out}(e)} (A_g', TS') \xrightarrow{\text{out}(e)} (A_g'', TS'') \xrightarrow{\text{out}(e)} (A_g''', TS''') \xrightarrow{\text{out}(e)} (A_g'''', TS''')$$

with $e$ matches $^c t$, then $A = B$. 


Proof: It is enough to consider the property \( \text{ProdAuth}(\langle A_\mathit{a}, \mathit{TS} \rangle, A, e) \) which states that \( A \) is the unique producer of \( e \) even if all the agents but \( A \) are malicious (i.e. their state correspond to \( S_E \)). Suppose, proceeding by contradiction, that \( B \neq A \). Consider now the agent \( B_{S_E}^{\phi_B} \) that can perform also all the action of the agent \( B_{S_B}^{\phi_B} \) of the initial system. Thus, by hypothesis, also this agent has the ability to execute (after the production of the entry \( e \)) an input/read operation \( o_p(t) \), with \( e \) matches \( t \), which returns \( \langle \hat{d} \rangle \). After the execution of this operation this agent surely has in its knowledge both \( c \) and \( \hat{d} \), hence it can produce the entry \( \langle \hat{d} \rangle^{[c]} = e \). The fact that \( B \neq A \) contradicts the hypothesis \( \text{ProdAuth}(\langle A_\mathit{a}, \mathit{TS} \rangle, A, e) \).

2.4 Asymmetric Partitioning of the Tuple Space

In the previous section we have proposed to logically partition the tuple space, in such a way that an agent can access a partition only if it is able to show a proof of knowledge of the partition field value. If an agent can access a partition, then it can perform write, read, and take operations on that partition.

We have formally proved that even of this approach can support secrecy, authenticity, and integrity, it is possible to support producer and receiver authentication only in the trivial case in which the producer and the consumer are the same agent.

This limitation of partition field is a consequence of the fact that they are not expressive enough to discriminate between the producers and the readers/consumers of a datum. As described in the proof of Proposition 2.2, when an agent is able to read an entry, then it is also able to reproduce that entry. The same holds also for producers; when an agent produces an entry, it can also read or consume that entry.

However, many applications need to support this form of discrimination. Consider, for example, a client-server service (such as a print service): clients send the requests and the server is the only agent that collects them. Using a tuple space to implement the model, the client submits the work writing an entry,
and the server consumes the requests. It is important to allow the permission to read the submitted jobs only to the server. Furthermore, several applications need to authenticate the sender or the receiver of a specific entry. For example, the distribution of software: a software house distributes its software products, upgrades and patches, writing them into the space. In order to avoid to download and install uncertified code, the receiver requires the authentication of the data producer.

In order to support this class of applications, the following section introduces entries with a new field of control, named asymmetric partition field.

2.4.1 Asymmetric partition fields

As commented above, we intend to use asymmetric partition fields to distinguish between write and read/input permissions, and to provide a manner for authenticating the sender ("who has written the entry"), and the receiver ("who is executing the read/input").

The idea is to consider a new class of partitions which have the following characteristics: given a partition, there exists two corresponding partition field values \( K \) and \( \overline{K} \), the former used to introduce data in the partition, and the latter used to retrieve data. In other words, given an entry containing the control information \( K \), an agent can access (read or remove) that entry only if it is able to show a proof of knowledge of \( \overline{K} \). In this way, if \( K \) is known only to the agent \( A \), and \( \overline{K} \) is known only to the agent \( B \), then the produced and the consumer of the entry are ensured to be \( A \) and \( B \), respectively.

Formally, let \( \text{APartition} \), ranged over by \( K, K', K_1, \ldots \), be a denumerable set of asymmetric partition fields. We also assume that \( \text{APartition} \) contains a special default value \( ? \), whose meaning is similar to \# for partition fields. Let \( \vdash : \text{APartition} \to \text{APartition} \) be a function such that if \( \overline{K} = K' \) then \( \overline{K'} = K \) (and \( \overline{?} = ? \)).

The idea is to exploit this new kind of fields in the matching rule, in order to capture a more sophisticated relation between entries and templates. In particu-
lar, we exploit the fact that the information that the agents must know in order to either read/input or write an entry are different. Hence, limiting the knowledge of this information, we can distinguish the write and the read/input permissions. An entry \( e \), and a template \( t \), with asymmetric partition (and partition) fields, are defined as follows:

\[
e = \langle \bar{d} \rangle_{[k]}^{[c]}
\]

\[
t = \langle \bar{d} \rangle_{[k]}^{[c]}
\]

Let \( \text{Entry}_K \) and \( \text{Template}_K \) be the set of entries and templates with asymmetric partition (and partition) fields, respectively. In the following, \( e, e', \ldots \), range over \( \text{Entry}_K \), and \( t, t', \ldots \), range over \( \text{Template}_K \). We define \( e|_d \) as the operator that, given an entry \( e = \langle \bar{d} \rangle_{[k]} \), returns the tuple of data \( \langle \bar{d} \rangle \).

**Definition 2.7 Matching rule with asymmetric partition fields** - Let \( e = \langle \bar{d} \rangle_{[k]}^{[c]} \) be an entry, and \( t = \langle \bar{d} \rangle_{[k]}^{[c]} \) be a template; we say that \( e \text{ matches}_K t \) if the following conditions hold:

- \( K = K_t = ? \) or \( \overline{K} = K_t \);

- \( \langle \bar{d} \rangle^{[c]} \text{ matches}^{c} \langle \bar{d} \rangle^{[c]} \).

The definition of \( \text{matches}_K \) is a conservative extension of \( \text{matches}^{c} \) (see Definition 2.5).

Observe that the asymmetric partition field \( ? \) is used as a default do-not-care field; it can be used, e.g., to produce and retrieve entries for which it is not important to show the knowledge of any specific asymmetric partition field. Therefore, for the sake of simplicity, the asymmetric partition field is sometimes omitted when it is \( ? \).

**Example 2.3 matches\(_K\) evaluation** - Let \( K \neq ? \) and \( K_t = \overline{K} \) be asymmetric partition fields. A list of entry matching examples follows.
The semantics of the primitives, using the entry (template) with asymmetric partition field and the matching rule matches\(_E_\), is obtained simply by replacing in Table 2.1 matches with matches\(_E_\), and the e, used to describe the return value, with e\(_{id}\) (in this context ReturnValue = \{e\(_{id}\) | e \(\in\) Entry\(_E\) \(\cup\) \{-\}\}). Observe that also in this case the asymmetric partition fields are used only to control the access to the entries, and the return value does not include them.

In this case, the fact that the additional control fields of an entry is not returned as the result of a read/consume operation is a crucial aspect. Consider e.g. an application in which we want to use the asymmetric partition field K to authenticate the entries produced by an agent with identity A. To this aim, it is fundamental that K is a secret known only by the agent A during the overall computation. The agent A will add the asymmetric partition field K to those data that will need to be authenticated. Suppose now that another agent reads/consumes the authenticated entry; in the case it receives also the information K, then K will become part of its knowledge, thus contrasting the expected secrecy of K.

The asymmetric partition field presents some similarities with the asymmetric field and object lock used by SecOS. However, differently from SecOS, our control fields are associated only to the entries (and not to the data fields) and, more important, the return value does not contain the control fields. A more detailed comparison is made in Section 2.6.

In order to model the agents under the new kind of control fields, we can adopt the rule in Table 2.2, simply by extending the set Knowledge that now must contain also the asymmetric partition fields, i.e. Knowledge = \(P(Data) \cup P(Partition) \cup P(APartition)\), and assuming that D(\(\phi\)) returns the set of entries.

<table>
<thead>
<tr>
<th>template</th>
<th>does it match with (d)(_{[K]})?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(null)(_{[K]})</td>
<td>no</td>
</tr>
<tr>
<td>(null)(_{[K]})</td>
<td>yes</td>
</tr>
<tr>
<td>(d)(_{[K]})</td>
<td>no, if (K' \neq K_t)</td>
</tr>
</tbody>
</table>
and templates that can be generated by using the data fields, the partition and the asymmetric partition fields in $\phi$ (plus null, $\#$, and ?).

Also in this case, we can assume the scope of the asymmetric partition field either static or dynamic. For the reason described in the Section 2.3.2, the dynamic approach is more flexible; for this reason, we consider the dynamic approach.

We are now ready to show that the addition of asymmetric partition fields is enough to support both producer and receiver authentication. Indeed, if we consider two asymmetric partition fields $K$ and $K'$, and we assume that $K$ is known only by the producer while $K'$ is known only by the receiver, then all the data exchanged using the asymmetric partition fields $K$ and $K'$ are authenticated. The following proposition defines sufficient conditions which implies authentication.

**Proposition 2.3** Let $(Ag, TS)$ be a system, $A_{S_A}^{\phi_A} B_{S_B}^{\phi_B} \in Ag$ be two agents. Let $K$ and $K' = K$ be two asymmetric partition fields. If the following conditions hold:

- i) $K \in \phi_A$;
- ii) $K' \in \phi_B$;
- iii) $Secrecy((Ag, TS), \phi_f, K)$ holds for $\phi_f = \bigcup_{e \in TS \text{ dataset}(e)} \bigcup_{C_S^{\phi} \in Ag, C \neq A} \Phi$;
- iv) $Secrecy((Ag, TS), \phi_f, K')$ holds for $\phi_f = \bigcup_{e \in TS \text{ dataset}(e)} \bigcup_{C_S^{\phi} \in Ag, C \neq B} \Phi$.

then for any entry $e$ with asymmetric partition field set to $K$, we have that $ProdAuth((Ag, TS), A, e)$ and $RecvAuth((Ag, TS), B, e)$ hold.

**Proof:** To prove producer authentication it is enough to observe that if during a computation an agent $C_S^{\phi} \in Ag$ performs out$(e)$, then the agent $C$ must have $e \in D(\phi)$, thus also $K \in \phi$. By iii) only $A$ can have $K$ in its knowledge during all the computation, thus $C = A$.

Similarly, to prove receiver authentication we simply observe that a template $t$ matching with $e$ must have asymmetric partition field set to $K'$, but iv) ensures that $K'$ can be in the knowledge of no agent but $B$. 

2.4.2 An implementation of asymmetric partition fields

We have described the asymmetric partition fields as a generic set of names equipped with a particular function $\tau$ which associate to each field $K$ the corresponding matching field $\overline{K}$.

There are two fundamental features that the function $\tau$ must satisfy. The former is that the coordination middleware (that is the implementation of the tuple space) should be reasonably able to check whether two fields $K$ and $K'$ match (i.e. verify if $K' = \overline{K}$). The latter regards the impossibility for an agent to guess $\overline{K}$ starting from the knowledge of $K$, more formally, any agent that has in its knowledge $K$, has no reasonable way to guess $\overline{K}$ until it explicitly reads this information from the tuple space.

The section presents a possible way to implement the asymmetric partition fields and the function $\tau$ exploiting asymmetric cryptography (see e.g. [Sch96]). To this end, we first give some formal definitions:

- The set of plaintexts $\text{PlainText}$, ranged over by $p, p', \ldots$;

- The set of encryption keys $\text{Key}$, ranged over by $k, k'$, $\text{PrivK}, \text{PubK}, \ldots$, containing private and public keys. As usually done when cryptographic protocols are analysed (see e.g. [DY83]), we assume a perfect version of cryptographic operations. Hence, in the following, when we refer to pairs of private and public keys $(\text{PrivK}, \text{PubK})$, we assume that a plaintext encrypted with $\text{PubK}$ (resp. $\text{PrivK}$) can be decrypted only using $\text{PrivK}$ (resp. $\text{PubK}$);

- The set $\text{Ciphertext}$, ranged over by $s, s', \ldots$, that contains the ciphertexts obtained encrypting $p \in \text{PlainText}$, with $k \in \text{Key}$, denoted with $[p]_k$.

Any non trivial asymmetric partition field, i.e. those different from $\tau$, is encoded as a triple $(p, \text{PubK}, s)$ composed of a plaintext $p$, one public key $\text{PubK}$ and one ciphertext $s$. Suppose that we have two asymmetric partition fields $(p, \text{PubK}, s)$ and $(p', \text{PubK}', s')$; two fields match if and only if the decryption with $\text{pubK}'$ of the ciphertext $s$ produces the plaintext $p'$ and, symmetrically, the decryption
with \( \text{pubK} \) of the ciphertext \( s' \) produces the plaintext \( p \). This condition holds if and only if \( s \) corresponds to \( [p']_{\text{PrivK'}} \) and \( s' \) corresponds to \( [p]_{\text{PrivK}} \).

More precisely, the implementation is as follows: the asymmetric partition field \( ? \) is implemented with a default value \( ?' \), while all the other fields are encoded as triples \( (p, \text{PubK}, s) \). The function \( \overline{?} \) is defined as follows:

- given \( ?' \), we have that \( \overline{?'} = ?' \);
- given the triple \( (p, \text{PubK}, s) \), we have that \( \overline{(p, \text{PubK}, s)} = (p', \text{PubK'}, s') \) if \( s = [p']_{\text{PrivK'}} \) and \( s' = [p]_{\text{PrivK}} \).

Observe that this implementation requires that, in order to produce an asymmetric partition field \( K \), an agent should know a private key \( \text{PrivK'} \) as well as the public key \( \text{PubK} \) corresponding to the private key \( \text{PrivK} \) used to produce \( \overline{K} \). In light of this observation, we can conclude that it is not reasonably possible for an agent to generate \( \overline{K} \) given only \( K \), because asymmetric cryptography ensure that it is not reasonably possible to produce the private key \( \text{PrivK} \), given the corresponding public key \( \text{PubK} \).

### 2.4.3 Examples

In order to prove the adequacy of asymmetric partition field to solve the security issues that we have tackled, this section presents several examples. In particular, we hint how to solve problems such as distinguishing the write and read permission, as well as the authentication of the sender and of the receiver of entries.

#### Write and Read/Input Permissions

Using asymmetric partition fields it is possible to avoid the unauthorized replication of entries, i.e. avoid that a reader/consumer of an entry is always able to reproduce that entry. This feature can be useful in several applications, e.g., to distinguish write and read/input permissions. On the contrary, if the readers
can write the entries that they read, this means that read permissions implies also write permissions.

Let \( K \) and \( K' = \overline{K} \) be asymmetric partition fields. Suppose that only agent \( A \) knows and keeps secret \( K \), whilst \( K' \) is known to all the other agents. Thus, we have that \( A \) can write an entry of the form \( \langle \tilde{d} \rangle_{[K]} \) (assuming that also \( \tilde{d} \) is in its knowledge) that any other agent can read (or consume). By definition, the return value of the read (or input) operation does not include the asymmetric partition field. Hence, the reader can subsequently produce a new entry with the same data fields \( \tilde{d} \), but a different asymmetric partition field. In this way, the new entry cannot be exactly the same of the initial one. Therefore, excluding the case when the reader is the sender, the reader cannot reintroduce the entry in the space.

Formally, let \( \{A,g,TS\} \) be a system in which \( A_g^\phi \in A.g, \langle \tilde{d} \rangle_{[K]} \in D(\phi) \) and \( TS = \{\langle K' \rangle, \langle \tilde{d} \rangle\} \). We suppose that \( K \) is known only by the agent \( A \); i.e. for any \( B_g^{\phi'} \in A.g \) with \( B \neq A \) then \( K \not\in \phi' \). We consider that the agent \( A \) simply produces the entry \( \langle \tilde{d} \rangle_{[K]} \), i.e. the unique actions in \( act \) starting from the state \( S \) is \( (S, \phi, out(\langle \tilde{d} \rangle_{[K]}), - , S_1) \) (where \( S_1 \) is a terminating state).

It is not difficult to prove that \( Secrecy(\{A_g, TS\}, \phi_E, d) \) holds for any \( \phi_E \) that does not contain \( K \) (i.e. \( K \not\in \phi_E \)). As a consequence of this fact we have that if \( \{A_g, TS\} \rightarrow^* \{A_g', TS'\} \) then we have that \( B = A \), because only \( A \) knows \( K \) (i.e., it is the unique agent with the necessary knowledge to set \( K \) as asymmetric partition field).

Symmetrically, one may wonder if write permissions include also read/input permission, that is, if an agent can always read/consumes the entries that it produces. The answer is negative, in particular, this is not permitted when an entry has \( K \) as asymmetric partition field and the producers does not know the corresponding \( \overline{K} \). For instance, all agents of the system described above have the ability to produce the entry \( e = \langle \tilde{d} \rangle_{[K]} \) because \( K' \) and \( \tilde{d} \) are public. However, only the agent \( A \) holds \( K \), thus it is the unique agent that can read \( e \).

Therefore, using the asymmetric partition field, we obtain the maximum flex-
bility to assign the write and read/input permissions simply by distributing, in the correct way, the knowledge of the asymmetric partition fields.

Secure channels

In the Example 2.2 we have shown that private partitions can be used to exchange data in such a way that secrecy, integrity, and availability are supported. Here, we simulate a secure channel can be simulated using the implementation of the asymmetric partition fields in term of asymmetric cryptography presented in Section 2.4.2. This encoding of secure channels guarantees also producer and receiver authentication.

We make the standard assumption that the involved agents have settled their private keys, and published the corresponding public keys. Let $P_{rivK_A}$ (resp. $P_{rivK_B}$) be the private key of $A$ (resp. $B$), and let $P_{pubK_A}$ (resp. $P_{pubK_B}$) be the public key of $A$ (resp. $B$). The emission of a datum $d$ on the secure channel can be performed by the agent $A$ simply by executing an $out([d]_{P_{pubK_B}}, [p]_{P_{privK_A}})$ operation. Symmetrically, a receive operation on the secure channel can be performed by $B$ simply executing an $in([null]_{P_{pubK_B}}, [p]_{P_{privK_B}})$ operation.

According to the discussion about the write and read/input permissions reported above, formalized in Proposition 2.3, we can clearly see that $A$ is the unique agent which has write capabilities on the channel, while $B$ is the unique agent with the input capability.

Differently from the channel implemented with private partition of Example 2.2, here we can exploit the different permissions to authenticate the producer and the consumer of a datum.

A consideration on $p$ is needed; $p$ is a plaintext on which $A$ and $B$ agrees in order to implement the secure channel. Different approaches could be followed in order to settle $p$. On the one hand, $p$ could be a public datum which is always used by $A$ and $B$ every time they need to communicate through the secure channel. On the other hand, $p$ could be refreshed during each communication sessions simply by, e.g., transmitting a new plaintexts $p'$ (to be replaced
to $p$) within each tuple communicated through the secure channel. The latter approach is inspired by the typical technique of nonces (numbers used only once) used in cryptographic protocols in order to ensure the freshness of the exchanged messages, thus contrasting the so-called reply attacks.

### 2.4.4 Partition/Asymmetric Partition Fields and Symmetric/Asymmetric Cryptography

Security protocols based on channel based communication, typically use symmetric and asymmetric cryptography, in order to guarantee security properties (e.g. secrecy, authentication). In this subsection we discuss interesting similarities between symmetric and asymmetric cryptography with the partition fields and asymmetric partition fields that we have introduced in the context of Linda-like coordination.

Suppose that the agents $A$ and $B$ in a cryptographic system are the unique agents who knows the symmetric key $K_{AB}$; then they are also the unique agents that can encrypt data with $K_{AB}$, as well as decrypt the obtained ciphertext. Similarly, if two agents $A$ and $B$ in our coordination system share (secretly) the partition $c_{AB}$, then only $A$ and $B$ can write, read, and consume any entry $e = (d)^{c_{AB}}$. Thus, we have that either solutions can be used to guarantee the secrecy of the data exchanged between $A$ and $B$. Moreover, in either solutions there is no way to authenticate who produces, or who receives, the data.

We have that asymmetric cryptography supports interesting extra functionalities: (i) signature of data in order to allow the receiver to authenticate the sender (by encrypting the data with the private key of the sender); (ii) emission of encrypted message, in such a way that it is guaranteed that only the expected receiver can decrypt it (by encrypting the data with the public key of the receiver). We have shown above that the same functionalities can be obtained in coordination infrastructures exploiting asymmetric partition fields.

A final remark regards a more detailed comparison between the coexistence in
our coordination model of both partition fields and asymmetric partition fields. It is not difficult to see that the former can be implemented in the latter. Let us consider, e.g., a system $(A, TS)$, and a partition field $c$; we encode this partition field using two related asymmetric partition fields, namely $K_c$ and $K_c'$. Given an agent $A^c_S ∈ A$ who knows $c$ (i.e. $c ∈ φ$), we replace $φ$ with $φ' = φ \{c\} \cup \{K_c, K_c'\}$. Any output operation $out(\langle \tilde{d} \rangle^{[c]}_\phi)$ on partition $c$ is encoded as $out(\langle \tilde{d} \rangle^{[K_c]}_{[K_c']_1})$, whilst any input operations $in(\langle \tilde{d} \rangle^{[c]}_\phi)$ is encoded as $in(\langle \tilde{d} \rangle^{[K_c]}_{[K_c']})$ (and similarly for read operations).

Even if the standard partition fields can be simply encoded in terms of the asymmetric partition fields, we continue to support both for two main reasons. The first one concerns efficiency: the verification of perfect matching between two partition fields is surely more efficient than the verification of correspondence of two partition fields $K$ and $K'$ (i.e. verify whether $K' = K$). For example, in the implementation described in Section 2.4.2 this latter check requires to use a decryption (or encryption) algorithm twice. A second interesting aspect is related to an interesting use that can be made of these two kinds of partition fields. For example, under public key infrastructures, a communication session could be based on (more efficient) symmetric keys, which are freshly generated before the session, and exchanged using (the less efficient) asymmetric keys. Also in our coordination infrastructure, asymmetric partition fields could be used in order to exchange newly generated partition fields, that can be used to share data during a specific session.

### 2.5 Destructive and Non-Destructive Retrieve Operations

There are examples of applications in which several (or all) agents can read some data, but only a subset of them can delete such data. For example, a person can publish some data (e.g. name, phone, e-mail, ..), writing an entry in the tuple
space. It is reasonable to assume that this information cannot be removed by any other person, excluding the owner (for example, in order to update the published data). To support this, it is necessary to distinguish between read and input permissions.

To accomplish this goal, the asymmetric partition fields are not sufficient. For example, if an agent \( \text{agent} \) has the ability to read a specific entry \( \text{entry} \), this means that the agent \( \text{agent} \) can generate a template \( \text{template} \) such that \( \text{entry} \) matches \( \text{template} \); this template can be used also for an input operation.

In this section we present how to extend the entries in order to discriminate among their readers and consumers. The idea is simply to use two groups of control fields, and use the first group for non-destructive (\( \text{non-destructive} \)) operations, and the second one for destructive (\( \text{destructive} \)) operations.

Formally, let \( \text{Entry}_k(\text{rd}, \text{in}) \) be the set, ranged over by \( e, e', \ldots \), of entries with read and input permissions, defined as follows:

\[
e = \langle d \rangle_{[K]_{\text{rd}}[\text{rd}']}_{[\text{in}]}
\]

We define \( e|_{\text{d}} \) as the operator that, given an entry \( e = \langle d \rangle_{[K]_{\text{rd}}[\text{rd}']}_{[\text{in}]} \), returns the tuple of data \( \langle d \rangle \).

The control fields with the subscript \( \text{rd} \) are used in the case of \( \text{rd} \) operations; those with subscript \( \text{in} \) are used in \( \text{in} \) operations. The template continues to be defined as in the previous section (that is \( t \in \text{Template}_c \)); indeed templates are used in operations \( \text{rd}(t) \) or \( \text{in}(t) \), and the indication of the kind of retrieve operation (either destructive or non-destructive) is inherited by the operation itself (either \( \text{in} \) or \( \text{rd} \), respectively).

**Definition 2.8 Matching rule with permissions** - Let \( \text{op} \in \{\text{rd}, \text{in}\} \) be the primitive on which the matching must be evaluated, \( t \) be a template and \( e = \langle d \rangle_{[K]_{\text{rd}}[\text{rd}']}_{[\text{in}]} \) be an entry; we define \( e \) matches \( \text{op} \) \( t \) as follows:

- \( \langle d \rangle_{[K]} \text{ matches } t \), if \( \text{op} = \text{rd} \).
- \( \langle d \rangle_{[K']} \text{ matches } t \), if \( \text{op} = \text{in} \).
In other words, $\text{matches}_k^c(\text{op})$ is reduced to $\text{matches}_k^c$ using only the control fields of $e$ corresponding to $\text{op}$.

It is trivial to prove that $\text{matches}_k^c(\text{op})$ is well defined; in both cases $\text{matches}_k^c$ is applied to entry $e \in \text{Entry}_k^c$ and template $t \in \text{Template}_k^c$.

The semantics of the primitives, using the entry with permission and the matching rule $\text{matches}_k^c(\text{op})$, is obtained simply by replacing in Table 2.1 $\text{matches}_k^c$ with $\text{matches}_k^c(\text{op})$, and the $e$ used to describe the return value with $e_{|d}$ (in this context $\text{ReturnValue} = \{e_{|d} | e \in \text{Entry}_k^c(\text{rd, in})\} \cup \{-\}$).

The calculus we have defined in this section comprises the standard partition fields, the asymmetric partition fields, and permits also the indication of two different policies for read and input operations. As it comprises all the three new mechanisms we propose, this is the calculus which formally define the coordination model $\text{SecSpaces}$.

2.5.1 Examples

It is interesting to observe that the combination of the possibility to discriminate between out and rd/in permission (supported by the asymmetric partition fields as discussed in Subsection 2.4.3) plus the possibility to discriminate between the rd and in permissions introduced in this section, provide the maximal flexibility in the definition of the whole spectrum of permissions.

The two examples in this section report two typical patterns of interactions: a one-to-many communication in which there is a producer and many possible consumers, and a many-to-many communication in which the receivers can only read the data, that can be removed only by the producers.

Example 2.4 Distribution of fresh partition names - In the Example 2.3.3 we have discussed about the problem of generating fresh partitions to be used as private partitions during a secure communication session. Here, we present the description of a possible solution to this problem based on a server that produces the fresh partition field identifiers.
The idea is that a trusted agent behaves as the server responsible for the generation of the fresh partition field identifiers. When a client agent in the system needs to use a fresh partition field, it simply consumes one of the identifiers produced by the server. Following this approach, it is important to guarantee the authentication of the server, in order to give guarantees to the clients that they have consumed an entry generated by the server. The other important issue is that two different agents should not access the same fresh partition identifier. This can be ensured by giving to the clients the input permission, and not the read permission to the tuples containing the fresh partition field identifiers.

The specification of the service is as follows. Let \( \langle A \rangle, Ts \rangle \) be a system, let \( K, K' \) and \( K_s, K_r \) be two pairs of asymmetric partition fields such as \( K' = \overline{K} \) and \( K_r = \overline{K_s} \). We assume that \( K, K' \) and \( K_s \) are known only to the server who keep them secret, while \( K_r \) is public.

Each fresh partition field \( c \) can be communicated by the server simply by producing the entry \( \langle c \rangle_{K_r} \langle K_s \rangle_{\text{in}} \). The clients can get a fresh partition field simply by performing the operation \( \text{in}(\langle \text{null} \rangle_{K_r}) \).

Observe that producer authentication is ensured by the fact that the server is the unique agent who knows \( K_s \). Moreover, the clients have only input permission on these entries and not rd permission because they are aware of \( K_r \) while they do not know \( K' \). Only the server (the unique agent who knows \( K' \)) has rd permissions.

**Example 2.5 Many-to-many communication** - In this example we show how to encode the many-to-many communication model, distinguishing the permission of writing, reading and consuming the entries. The model we present extends the secure group communication discussed in the Example 2.2 by adding an access control policy that discriminates between the read, remove and output permission.

Let \( \langle A \rangle, Ts \rangle \) be a system, \( K_r \) and \( K_{wr} = \overline{K_r}, K_{wl} \) and \( K_l = \overline{K_{wl}} \) be two pairs of asymmetric partition fields. The partition fields \( K_r \) and \( K_l \) are used to access the entries during read and input operations, respectively (e.g. \( \text{rd}(\langle \text{null} \rangle_{K_r}) \) and
in(\langle \text{null}|k_i \rangle). On the other hand, \(K_{w_r}\) and \(K_{w_i}\) are used to produce the entries (e.g. \(\text{out}(\langle d|K_{w_r}|d|K_{w_i}|ln \rangle)).

We assume that the agents in \(A_g\) are divided into three groups: \(A_{g_w} \subseteq A_g\) is the group of writers; \(A_{g_r} \subseteq A_g\) is the group of agents that can perform only \(rd\) operations; \(A_{g_i} \subseteq A_g\) is the group of agents that can perform only in operations. The partition fields identifiers are distributed as follows:

i) if \(K_{w_r}\) (resp. \(K_{w_i}\)) is in \(\Phi\) then also \(K_{w_l}\) (resp. \(K_{w_r}\)) is in \(\Phi\) and \(A_{g_w}^{\Phi} \in A_{g_w}^{r}\);

ii) if \(K_r\) is in \(\Phi\) then \(A_{g_r}^{\Phi} \in A_{g_r}^{r}\);

iii) if \(K_i\) is in \(\Phi\) then \(A_{g_i}^{\Phi} \in A_{g_i}^{r}\).

We assume also that these asymmetric partition fields identifiers are not present in the initial tuple space TS (i.e. \(K_r, K_{w_r}, K_{w_l}, K_i \not\in \bigcup_{e \in TS} \text{dataset}(e)\)).

To be as general as possible, without specifying the whole system, we simply assume that agents do not distribute the access permissions \(K_r, K_{w_r}, K_i\) and \(K_{w_l}\) and use the correctly asymmetric partition fields, that is:

- for any \((A_g', TS')\) such that \((A_g, TS) \rightarrow^* (A_g', TS')\) then \(K_r, K_{w_r}, K_{w_l}, K_i \not\in \text{dataset}(e), K = K_{w_r}\) and \(K' = K_{w_l}\);

- for any \((A_g', TS')\) such that \((A_g, TS) \rightarrow^* (A_g', TS')\) then \(\text{if } op = rd \text{ then } K = K_r\) if \(op = in\) then \(K = K_i\).

Let us consider the system with the enemy; if \(\Phi_E\) is a knowledge set such that \(K_r, K_{w_r}, K_i, K_{w_l} \not\in \Phi_E\); the system \((A_g \cup E_{S_E}^{\Phi_E}, TS)\) is such that:

- only agents in \(A_{g_w}\) can write entries;

- only agents in \(A_{g_r}\) can read (without consuming) entries;

- only agents in \(A_{g_i}\) can consume entries.
Chapter 2. Security in tuple space coordination model

The approach we have followed is similar to the encoding of the secure channel (see Subsection 2.4.3). However, the assumption that asymmetric partition fields $K_r$, $K_i$, $K_w$, and $K_{wi}$ are known by more than one agent does not allow to authenticate the agents that write, read or consume the entries. In the special case in which $A_{g_w}$ is composed by one agent only, the authentication of the producer of an entry holds. Similarly, if $A_{g_r}$ (resp. $A_{g_i}$) is composed by one agent only, the authentication of the reader (resp. consumer) holds. The proof of these properties is essentially the same as the one given in Proposition 2.3.

2.6 Related work

In the Introduction we have already discussed two proposals, KLAIM and SecOS, that aim to solve the problem of adding access control policies to the primitives of Linda.

Here we want to add one further observation. The symmetric and asymmetric field locks of SecOS are used to select specific fields inside entries, hence they provide also a way to assign the permission for accessing the fields. Differently, in SecSpaces control fields limit the access to the entries. However, this does not prevent the possibility of limiting the access to a specific field of an entry. Let $d_i$ be a field contained in the entry $e = \langle d_1, \ldots, d_n \rangle_{[K]_r[K']_w}$. In order to restrict furtherly the access to $d_i$, we can proceed simply by splitting $e$ into two entries. The first one is $e' = \langle d_1, \ldots, d_{i-1}, d_e, d_{i+1}, \ldots, d_n \rangle_{[K]_r[K']_w}$ where $d_e$ is a fresh datum used as a pointer to the second entry $e'' = \langle d_e, d_i \rangle_{[K]_r[K']_w}$. Agents that can read (resp. remove) $e'$ are exactly those that can read (resp. remove) $e$. On the other hand, in order to read (resp. remove) the field $d_i$ stored inside $e''$ agents must know the pointer $d_e$ and the partition field $a$ (resp. $a'$). Thus, we can exploit $a$ (resp. $a'$) to restrict the access to $d_i$. If necessary, instead of partition fields $a$ and $a'$, it is possible to use also asymmetric partition fields in order to control the access to $d_i$. Using this idea, we can repeatedly split the entry until we obtain a distinct access permission for any data field.
Finally, some comments on the different definition of return value follow. We have already observed that in SecOS an agent able to read an entry is also able to reproduce that entry. In SecOS the return value of a read (or input) operation is exactly the matching entry that has been found in the space, hence it contains both data fields and fields/objects locks. In this way, the knowledge of the agent contains all the fields, even if some of them are inaccessible. Thus, malicious agents may use these chunks of data even if they do not know the information they contain. On the other hand, in SecSpaces when an agent accesses an entry, the return value does not contain the control fields. In this way, it is possible to avoid the possibility for the reading agent to use these information in subsequent operations.

Here we report a list of other significant proposals present in the literature that consider security issues in the setting of tuple space coordination. In [HR03] a password-based mechanism is proposed to protect the access either at the level of tuple space and single tuple. Access permission on entries can be associated at the read as well as the removal operations. Differently from SecSpaces, the in permission inherits the rd permission and, as in the examples reported in Section 2.5.1, this fact can be an undesiderable feature. Two proposals that consider access rights as in KLAIM are [MMU01] and [COZ00]. In [MMU01] the permission policies (laws) are described using Prolog rules; the agents have the correct access rights if they satisfy the rules. This check is executed by policy-independent trusted controllers. In [COZ00] the relation between coordination and access control in open, distributed agents systems is investigated in the context of the TuCSoN [OZ99] coordination model. More precisely, the possibility to program the coordination medium and the hierarchical topology supported by TuCSoN are exploited to support an access control to the shared resources, which is expressed in terms of an access control matrix. On the other hand, in [Pin92] an access control policy is defined which is similar to the one supported by SecOS. The idea is to provide Linda with directed communication via private channels. The solution exploits asymmetric cryptography: at each entry corresponds a spe-
cial data, the ticket. The matching is available only if the tickets of entry and template match (i.e. one of them is the public ticket and the other one is the corresponding private ticket).

A final remark regards the fact that we support the definition of the access control policies by introducing extra information inside the tuples. These extra information are usually called attributes, and they can be used not only to support secure coordination but also in many other contexts. A survey of the possibilities provided by the attributes can be found in [Woo99] where several manners to introduce the attributes in tuple space coordination are discussed, in particular how to insert them to the level of fields, of entries and of tuple spaces.

Most inherent 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 [BN03]: we could extend the syntax of data field by adding encrypted data, i.e. \( d ::= \ldots | [d]_{E_c} \), 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.

## 2.7 Conclusion

In this chapter we have proposed SecSpaces, a Linda-like coordination model that permits to control the access to the entries stored in the shared data space, and to authenticate/identify the producer of an entry or its reader/consumer. SecSpaces essentially introduces a new advanced matching rule, which exploits two new kinds of fields, namely partition and asymmetric partition fields.

In order to be suitable for open applications, in SecSpaces all the agents can always access the tuple space using the coordination primitives, in other words, SecSpaces permits to define permissions at the level of tuples and not at the
level of tuple space. For example, an agent can always introduce entries in the tuple space. In this way, e.g., the quality of the coordination service cannot be guaranteed: an agent, indeed, can write a huge amount of entries into the space degrading the performance of the coordination service itself. As discussed in the introduction, we consider the definition of access control policies at the level of tuple spaces an orthogonal task, and we leave this as a future work.

We want to conclude by remarking that we have considered a subset of the common Linda primitives. More precisely, we have not taken into account the non-blocking versions of `rd` and `in` operations, usually denoted as `rdp` and `inp`, respectively. The analysis of an extended model including these operations is left as future work. Actually, we feel that non-blocking operations increase the power of the enemy, because in some cases they could be used to test not only the presence, but also the absence, of a particular entry.
Chapter 3

A framework for checking security properties in SecSpaces

In the previous chapter we have investigated the main security issues emerging when a Linda-like language is exploited to coordinate systems open to untrusted agents. Then, SecSpaces have been proposed to support the applications for open systems and its adequacy has been motivated by describing some protocols to use for guaranteeing some security properties. In this chapter our efforts are addressed with particular care at the problem of formalizing SecSpaces. More precisely, in this chapter 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. The testing equivalence will be exploited to formalize -in general terms, i.e. without taking a particular system- some of the main security properties.

3.1 Introduction

In Chapter 2 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 chapter 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. an extended tuple structure and a matching rule 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 chapter is structured as follows. Section 3.2 gives the syntax of system descriptions, Section 3.3 presents the operational semantics of SecSpaces and Section 3.4 defines testing equivalence that we exploit to formally prove basic safety properties. Section 3.5 presents the formal techniques to be used for checking the various kinds of security properties against system descriptions. Finally, Section 3.6 comments related works and concludes the chapter.

### 3.2 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.
3.2.1 Entries and templates

Linda tuples are ordered and finite sequences of typed fields. SecSpaces extends the classical Linda tuple by adding special control fields, namely partition and asymmetric partition fields. Further, for the sake of simplicity, in the model we are going to present, 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 \( m, n, \ldots \), be an infinite set of messages, \( \text{Partition} \), ranged over by \( c, c_r, c_s, \ldots \), be the set of partitions. We also assume that \( \text{Partition} \) contains a special default value, say \( \# \), whose meaning will be discussed in the following. Let \( \text{APartition} \), ranged over by \( k, k', k_r, \ldots \), be the set of asymmetric partitions. Similarly to partition fields, we assume that \( \text{APartition} \) contains a special default value denoted by “?” as well. Let \( \overline{\text{APartition}} : \text{APartition} \rightarrow \text{APartition} \) be a function such that if \( \overline{k} = k' \) then \( k'' = k \) (and \( ? = ? \)). Let \( \text{Var} \), ranged over by \( x, y, \ldots \), be the set of data variables. We use \( \bar{x}, \bar{y}, \ldots \), to denote finite sequences \( x_1;x_2;\ldots; x_n \) of data variables.
The set $\text{Entry}$ of entries, ranged over by $e, e', \ldots$, is defined as follows:

$$e = \langle \tilde{d}\rangle_{[k]_r,d',k'_l}$$

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

$$\tilde{d} ::= d \mid d; \tilde{d}$$

$$d ::= m \mid c \mid k \mid x$$

and $c, c' \in \text{Partition}, k, k' \in A\text{Partition}$.

A data field $d$ can be a message, a partition, an asymmetric partition or a variable. In the following, we use $\text{Data}$ 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 $e = \langle \tilde{d}\rangle_{[k]_r,d',k'_l}$, $\tilde{e} = \tilde{d}$).

The set $\text{Template}$ of templates, ranged over by $t, t', \ldots$, is defined as follows:

$$t = \langle \tilde{d}t\rangle_{[k_l]}$$

where $\tilde{d}t$ is defined by the following grammar:

$$\tilde{d}t ::= dt \mid dt; \tilde{d}t$$

$$\tilde{d}t ::= d \mid \text{null}$$

and $c_l \in \text{Partition}, k_l \in A\text{Partition}$.

Differently from entries, data fields of templates can be set to wildcard value, denoted by $\text{null}$. 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 $t = \langle \tilde{d}t\rangle_{[k]}$ we write $t = \langle \tilde{d}t\rangle_{[k]}$) 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 $e = \langle \tilde{d}\rangle_{[c_l]}$ we simply write $e = \langle \tilde{d}\rangle_{[c]}$).
3.2.2 Programs

Here we formalize systems configurations, that include processes exploiting the 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 $A, B, \ldots$, and processes, ranged over by $P, Q, \ldots$, are defined as follows:

$$A, B, \ldots ::= \begin{align*}
&\text{systems} \\
&e & \text{entries} \\
&P & \text{processes} \\
&A | A & \text{parallel composition}
\end{align*}$$

$$P, Q, \ldots ::= \begin{align*}
&\text{processes} \\
&0 & \text{null process} \\
&\text{out } e.P & \text{output} \\
&\text{rd } t(\bar{x}).P & \text{read} \\
&\text{in } t(\bar{x}).P & \text{input} \\
&P | P & \text{parallel composition} \\
&!P & \text{replication}
\end{align*}$$

A system can be an entry, a process or the parallel composition of entries and processes. A process can be a terminated program ($0$), 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) $\text{out } e$, that writes the entry $e$ in the TS; ii) $\text{rd } t(\bar{x})$, that given a template $t$ reads a matching entry $e$ in the TS and stores the return value in $\bar{x}$; iii) $\text{in } t(\bar{x})$, that given a template $t$ consumes a matching entry $e$ in the TS and stores the return value in $\bar{x}$. A process $P | Q$ 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 \( P[\bar{d}/\bar{x}] \) to denote the process that behaves as P in which all occurrences of \( x \) are replaced with \( \bar{d} \). We also use \( P[\bar{d}/\bar{x}] \) to denote the process obtained by replacing in P all occurrences of variables in \( \bar{x} \) with the corresponding value in \( \bar{d} \), that is \( P[d_1;\ldots;d_n/x_1;\ldots;x_n] = P[d_1/x_1]\ldots[d_n/x_n] \).

We say that a system is well formed if each \( \bar{x} \) operation is such that the variables \( \bar{x} \) and the tuple of data \( \bar{d} \) have the same arity. Let \( \text{fn}(A) \) and \( \text{fv}(A) \) 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 \( \text{Sys} \) 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 \( \text{rd} \) operations and the other one to \( \text{in} \) operations. The definition of the matching rule follows.

**Definition 3.1 Matching rule** - Let \( e = \langle d_1; d_2; \ldots; d_n \rangle^{[c]}_{[k]} \) be an entry and \( \text{op} \in \{ \text{rd}, \text{in} \} \) be the operation using the template \( t = \langle dt_1; dt_2; \ldots; dt_m \rangle^{[c]}_{[k]} \). Let \( c_e \) and \( k_e \) be the control fields of \( e \) associated to \( \text{op} \), we define \( e \) matches \( t \) as follows:

1. \( m = n \)
2. \( dt_i = d_i \) or \( dt_i = \text{null} \), \( 1 \leq i \leq n \)
3. \( c_e = c_t \)
4. \( k_e = k_t \).

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.

### 3.3 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 3.1 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), \ldots, (viii).

\[
\begin{array}{ll}
(\text{i}) & A \equiv A \\
(\text{ii}) & \frac{B \equiv A}{A \equiv B} \\
(\text{iii}) & \frac{A \equiv B \quad B \equiv C}{A \equiv C} \\
(\text{iv}) & A \upharpoonright B \equiv B \upharpoonright A \\
(\text{v}) & (A \upharpoonright B) \upharpoonright C \equiv A \upharpoonright (B \upharpoonright C) \\
(\text{vi}) & \frac{A \equiv A'}{A \upharpoonright B \equiv A' \upharpoonright B} \\
(\text{vii}) & A \upharpoonright 0 \equiv A \\
(\text{viii}) & !P \equiv P \upharpoonright !P
\end{array}
\]

Table 3.1: Structural equivalence
Formally, let \( \text{Act} = \{ \tau \} \cup \{ e \mid e \in \text{Entry} \} \cup \{ t \mid t \in \text{Template} \} \cup \{ \epsilon \mid t \in \text{Template} \} \), ranged over by \( \alpha \), be the set of possible actions: \( \tau \) represents an internal action, \( e \) the output of the entry \( e \), whilst \( t \) and \( \epsilon \) represent a \( \text{in} \) and a \( \text{rd} \) operation using template \( t \), respectively. Let \( \text{RetV} = \{ \epsilon \mid e \in \text{Entry} \} \cup \{ - \} \), 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 \( \text{in} / \text{rd} \) operations.

\[
\begin{align*}
(1) \quad \text{out } e.P & \quad \frac{\tau}{\rightarrow} \quad e|P & \quad (2) \quad \text{in } t[\epsilon].P & \quad \frac{t}{\rightarrow} \quad P[\epsilon/x] \\
& \quad - & \quad - \\
(3) \quad \text{rd } t(\epsilon).P & \quad \frac{\epsilon}{\rightarrow} \quad P[\epsilon/x] & \quad (4) \quad e & \quad \frac{0}{\rightarrow} & \quad \epsilon
\end{align*}
\]

\[A \rightarrow A', \quad B \rightarrow B' \quad \text{e matches}_{\text{in}} \quad t \]

\[
\begin{align*}
& (5) \quad \frac{\tau}{\rightarrow} \quad A|B & \quad \frac{\epsilon}{\rightarrow} \quad A'|B' \\
& \quad - & \quad - \\
& \quad \epsilon & \quad A \rightarrow A', \quad B \rightarrow B' \quad \text{e matches}_{\text{rd}} \quad t
\end{align*}
\]

\[
\begin{align*}
& (6) \quad \frac{\tau}{\rightarrow} \quad A|B & \quad \frac{\epsilon}{\rightarrow} \quad A|B' \\
& \quad - & \quad - \\
& \quad \alpha & \quad A \rightarrow A' \\
& \quad \beta & \quad - \\
& \quad \beta & \quad A \equiv B, \quad B \rightarrow B' \quad A' \equiv B'
\end{align*}
\]

\[
\begin{align*}
& (7) \quad \frac{\alpha}{\rightarrow} \quad A | B & \quad \frac{\beta}{\rightarrow} \quad A' | B \\
& \quad - & \quad - \\
& \quad \beta & \quad A \rightarrow A'
\end{align*}
\]

**Table 3.2:** Operational semantics
The labeled transition system we use is a quadruple $(\text{System}, \text{Act}, \text{RetV}, \rightarrow)$ where $\rightarrow \subseteq \text{System} \times \text{Act} \times \text{RetV} \times \text{System}$. $(A, \alpha, \beta, A') \rightarrow$ (also denoted as $A \rightarrow A'$) means that the system $A$ can execute action $\alpha$ with return value $\beta$ and it evolves in the system $A'$. Table 3.2 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$(\bar{e}).P$ and rd$\ t(\bar{x}).P$ perform an input and read operation, respectively: when a matching entry $e$ is found in the TS the return value is $\bar{e}$ and the process behaves as $P[\bar{e}/\bar{x}]$. The consumption of an entry $e$ in the TS is represented in (4), whilst rule (5) describes that if a process performs an in$\ t(\bar{x})$ 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, whilst (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 $A \rightarrow A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_{n-1} \rightarrow A'$. In other words, $A \rightarrow A'$.

### 3.4 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 [FGM00]. 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 $\omega \in \text{Data}$ 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 $A \equiv (d_1;d_2;\ldots;d_n)\underbrace{|[^{\#}]}_{\omega}B$, 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 $\phi_E \subseteq \text{Data}$ be the knowledge set of the external environment, the set $E(\phi_E) = \{A \mid f\text{n}(A) \subseteq \phi_E\}$ represents the external environment (set of tests) we consider in the equivalence evaluation.

**Definition 3.2 Testing equivalence** - Let $A$ and $B$ be two systems and $\phi_E$ be the knowledge of the external environment; we say that $A \leq_{\text{may}} B$ iff for every test $(T,\omega)$ with $T \in E(\phi_E)$: $(A | T) \Downarrow \omega$ implies $(B | T) \Downarrow \omega$. $A$ and $B$ are testing equivalent (denoted by $A \approx B$) iff $A \leq_{\text{may}} B$ and $B \leq_{\text{may}} A$.

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.

3.5 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.

3.5.1 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 \( A(d) = \text{out}((d)^{[c]}).0 \) be the process that writes the entry \( (d)^{[c]} \) and then terminates, \( B = \text{in}(\text{null})^{[c]}(x).0 \) 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.

### 3.5.2 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 $P(d) = \text{out}(\langle d \rangle_{k})$ be the process that writes the entry $\langle d \rangle_{k}$ into the TS, and $R = \text{in}(\langle \text{null} \rangle_{\Phi}(x).F(x))$ 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. $k \not\in \text{fn}(R)$ and $k \not\in \Phi$, 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: $S_{\text{id}}(d) = P(d)|R_{\text{id}}(d)$, where $R_{\text{id}}(d) = \text{in}(\langle d \rangle_{\Phi}(x).F(d))$; ii) we test if the ideal system $S_{\text{id}}(d)$ 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 $R_{\text{id}}$ 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 $\Phi$, 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 $S_{\text{id}}(d) \approx S(d)$, 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 $\Phi$ 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 \( R \not\in \varphi_E \).

### 3.5.3 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 [FGM04, 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 \( S_{id}(d) = P(d)\|_{\text{in}} (\text{null})|_{R}(x).F(d) \) 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.

### 3.5.4 A simple protocol for secrecy and authentication

The protocol proposed in Section 3.5.1 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 K. Let A = out((c)k). in (null)c(x). 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)d(x).out((d)k). 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 φE, with c, k, K ∈ φE, 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) ≈ 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.

3.5.5 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 \textit{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 \textit{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 \textit{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 “\textit{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 \(A = \text{out}(\langle d \rangle[^e_{\text{in}}]_{\text{e}'})\).P be the process that writes the entry \(\langle d \rangle[^e_{\text{in}}]_{\text{e}'}\) and then behaves as P; if only A knows \(e'\) (i.e., \(e' \notin \phi_E\)) and it keeps \(e'\) secret (i.e., secrecy of \(e'\) 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 \(e\) to that set of processes.
3.6 Conclusion

In this chapter 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.

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).
Chapter 4

Supporting Web Services Coordination with SecSpaces

Web Services standards and protocols (WSDL, UDDI, SOAP, etc.) are the basis of a novel technology supporting Web based applications. The basic elements of this infrastructure are the Web Services; these are components offering ports at which service invocations can be sent using XML-based protocols. The tools currently proposed for specifying and programming the interdependencies among Web Services (BPEL, BizTalk, etc.) support the description of the flow of service invocation needed among collaborating Web Services in order to complete a specific task. In this chapter we discuss the design and the implementation of a higher-level interaction model for Web Services that follows the tradition of data-driven coordination: Web Services do not collaborate via direct service invocation, but their interaction is mediated by a coordination space where shared data are stored and retrieved. Moreover, our proposal exploits the SecSpaces coordination primitives that, by exploiting a more sophisticated pattern matching mechanism, supports a controlled access to the shared data.

4.1 Introduction

Web Services technology tends to provide standard mechanisms for describing the interface and the services available on the web, as well as protocols for lo-
cating such services and invoking them (see e.g. WSDL [CCMW], UDDI [UDD], SOAP [BEK⁺]). More precisely,

- Web Services Description Languages (WSDL) is a format for describing network services as a set of endpoints operating on messages;

- Universal Description, Discovery and Integration (UDDI) serves as a discovery service for the WSDL descriptions;

- Simple Object Access Protocol (SOAP) is a simple and lightweight mechanism for creating the messages exchanged among the Web Services.

A relevant feature in the context of Web Services is the support for composing them in order to achieve complex tasks. For example, in business-to-business processes it is often the case to define new processes out of finer-grained subtasks that are likely available as Web Services.

The main effort in this respect is devoted to the definition of extensions of the Web Service standards, that support the definition of complex services out of simpler ones – the so called Web Services choreography. Several proposals that describe Web Services choreography have been already set up: BPML [BPM] by BPMI.org, XLANG [Tha] and BizTalk [Cor] (a visual specification environment for XLANG) by Microsoft, WSFL [Ley] by IBM, BPEL4WS [CGK⁺02] by a consortium grouping BEA, IBM, Microsoft, and others), etc. These proposals permit to specify the interdependencies among collaborating Web Services via the indication of the flow of their relative invocations. Figure 4.1 describes the layers composing the current Web service technologies.

In this chapter we discuss the design and the implementation of a higher-level interaction model for Web Services that follows the tradition of data-driven coordination: Web Services do not collaborate via direct service invocation, but their interaction is mediated by a coordination space where shared data are stored and retrieved.

The main advantage following from the introduction of a coordination space is loose coupling. In standard web based applications we have, on the contrary,
tight coupling in at least two ways. The first is that collaborating services must be up and running in order to complete their interaction. The second is that the interaction is programmed based on the interface of the services themselves; if the interfaces change, the application may break.

By loose coupling we mean that a Web Service can broadcast a request and then terminate. Some time later, a service may come online and may serve the request. This is said to be loose coupling in both “time and space”: the interacting services do not need to be connected simultaneously and the application do not need to know the actual location of a service at the time the service is required. Moreover, since the communications do not occur directly but are mediated by the coordination space, the applications do not need to be programmed on the basis of the interfaces of the collaborating services. In this way, the modification/update of the interfaces does not influence the designed application.

While trying to introduce data-driven coordination in the Web Services architecture we have tried to be as much as possible compliant and conservative with the architecture itself. For this reason we have designed a Web Service, that we have called WSSecSpaces, which offers the coordination primitives as invocable services. In this way, for a Web Service is sufficient a direct connection to a WSSecSpaces in order to coordinate its activity with the other collaborating Web
The coordination facilities of WSSecSpaces are essentially those offered by the generative communication paradigm [Gel85]: a sender communicates with one or more receivers through a shared space where the emitted messages are collected; a receiver can read or consume the message from the space indicating with a pattern the kind of message he is interested in. This form of communication is referred to as generative communication because when a message is produced, it has an existence which is independent of its producer, and it is equally accessible to all components. Figure 4.2 depicts the context in which coordination languages should be placed among the current Web service technologies: i) choreography languages can exploit a coordination platform in order to program the collaboration among Web services, ii) Web service registries and service discovery protocols can be based on a coordination platform (Chapter 7 implements these services by exploiting an extension of WSSecSpaces), and iii) since the WSSecSpaces is a Web service and it is used to allow the collaboration among Web services, it deals with the Web service description layer.

The main drawback of this approach, when applied to open applications such as those based on the Web, is that also untrusted and malicious components may access the coordination space. For this reason, we extend the coordination fa-
ilities of WSSecSpaces with a sophisticated pattern matching mechanism that permits to restrict the visibility and the accessibility, as well as verifying the producer, of a specific message inside the space. This extended pattern matching mechanism is borrowed from SecSpaces presented in Chapter 2, a data-driven coordination model designed for supporting security properties also in the context of open applications.

This chapter is structured as follows. In Section 4.2 we discuss the choices behind the implementation of WSSecSpaces; Section 4.3 discusses the related literature while Section 4.4 reports some conclusive remarks.

4.2 The WSSecSpaces Web Service

WSSecSpaces is a Web service that implements the coordination primitives of SecSpaces. In order to show that WSSecSpaces provides a coordination platform supporting some mechanisms for the secure collaboration among agents in presence of untrusted ones, we refer to the security issues described in Section 1.3 and for each one we discuss the adopted solution. We have already proved in chapters 2 and 3 that the SecSpaces coordination model supports some forms of control of the accesses to the entries stored in the shared space; in this section we investigate the implementation issues of SecSpaces and we show that Web Services technologies are an ideal platform for a secure implementation of SecSpaces. Finally, we describe how control fields are implemented (Section 4.2.1), how to represent entries and templates, the matching rule we adopt (Section 4.2.2) and some implementation details (e.g., WSDL) of WSSecSpaces (Section 4.2.3).

As outlined in Section 1.3, the second security problem concerns the communication on channels used by clients to interact with the Space Plugs. Such problems, regarding entity authentication, privacy and integrity of data, are well known problems and several cryptographic protocol [Sch96] have been proposed to solve them. In order to avoid such attacks, our implementation exploits the
HTTPS protocol between clients (e.g., other Web services) and WSSecSpaces. Finally, as will be explained in detail in Section 4.2.3, WSSecSpaces provides just one access port that can be used to perform output, read and removal operations on the space. In other words, our proposal has only one space plug that is described by the WSDL (Web Services Description Language [CCMW]) document associated with WSSecSpaces. WSDL documents are usually published on a UDDI server; the problem of obtaining a trusted reference at the document is left to the interaction phase between clients and the UDDI server.

4.2.1 Implementation of control fields

In this section we tackle with particular care the problem of implementing the SecSpaces control fields modeled in Chapter 2. The implementation of partition fields does not involve relevant problems; indeed, the only assumption is that a process should not have any way to guess an unknown partition field used by other processes. Similarly to symmetric encryption keys (see e.g. [Sch96]), we need to implement the set Partition so that to guess one of its values has low probability that can be realized, e.g., by encoding partitions with data composed by 512 bits.

The implementation of asymmetric partition fields is more sophisticated because the function \( \pi \) must guarantee that: i) the coordination service (that is the implementation of the shared space) should be able to check whether two fields \( k \) and \( k' \) match (i.e. verify if \( k' = \pi k \)); ii) it should be unfeasible for an agent to guess \( k \) starting from the knowledge of \( k' \). The implementation we use is that described in Section 2.4.2. In a few words, it exploits asymmetric cryptography: each asymmetric partition, except the default value \( ? \) encoded with \( ?' \), is encoded by a triple \((p, PubK, s)\) where \( p \) is a plaintext, \( PubK \) is a public key and \( s \) is a ciphertexts (i.e. a text obtained by encrypting a plaintext). Let \( \text{decrypt}(s, k) \) be the function that, given a ciphertext \( s \) and a cryptographic key \( k \), returns the plaintext obtained by decrypting \( s \) with key \( k \); the \( \pi \) is defined as follows:
4.2.2 Entry, template and XML-based matching rule

The SOAP protocol [BEK+] – used to perform Web services invocations – as well as most of the data involved in the Web Services technology are based on XML: this simplifies and enables interoperability between applications running on different architectures [Bos97]. In order to be consistent with these goals, WSSecSpaces provides a definition of entry and template based on XML. In the following we describe how we have defined the basic elements of SecSpaces, i.e., entries, templates and the matching rule in WSSecSpaces.

The structure of entries and templates has been defined by exploiting XML-Schema [Con]; the XML document (file secS.xsd) is reported in Appendix A.1.

An entry is described by an element entry which contains three elements: rd, in and datafield. The rd and in elements represent the control fields associated with rd and in operations, respectively. Their content is composed by two elements: apartition and partition. The former represents the asymmetric partition field and – following the implementation of Section 4.2.1 – contains three elements: plaintext, key and ciphertext that contain a plaintext (of type “string”), a cryptographic key (of type “string”) and a ciphertext (of type “binary”), respectively. The partition field is the content of partition (of type “string”). The third element of entry is dataField and contains the sequence of data fields. More precisely, it contains an unbounded sequence of field elements that can contain any XML type. An example of entry is reported in Table 4.1.

Differently from entries, templates contain only one occurrence of control fields and their data fields can also be set to wildcard. Templates are represented by the element template that contains three elements: apartition, partition (which have the same meaning and structure of the homonymous elements contained in
Sauron, the Dark Lord, has gathered to him the Rings of Power.

Table 4.1: Entry in XML
Sauron, the Dark Lord, has gathered to him the Rings of Power

48-1261-7348-1261446742-12272-50564148-1261312-127-1270-312783-12729117184182-


<template>
  <apartition>
    <plaintext>Sauron, the Dark Lord, has gathered to him the Rings of Power</plaintext>
    <key>48-1261-7348-1261446742-12272-50564148-1261312-127-1270-312783-12729117184182-...</key>
    <apartition>4-9x7cvf</apartition>
  </apartition>
  <tdataField>
    <field>
      <name>Roberto Lucchi</name>
    </field>
    <wnull/>
  </tdataField>
</template>

Table 4.2: Template in XML
entry) and tdataField. The latter one contains an unbounded sequence composed by field, wfield and wnull elements. The elements field have the same meaning of the homonymous ones contained in entry and the matching between two field elements holds if they have the same content. wfield and wnull are used to implement wildcards. Similarly to recent object oriented Java coordination middlewares such as TSpaces [WMF98] and JavaSpaces [Sun02] that allow the use of complex classes as typed wildcards, WSSecSpaces improves the matching rule by exploiting XML technologies. More precisely, in WSSecSpaces the elements wfield are used to specify an XML-Schema: an element field matches with an element wfield if its content can be validated with the schema contained in the latter element (i.e., it is conformant with the structure defined in the wildcard schema).

Finally, the element wnull (whose content is empty) provides a more general wildcard implementing the null defined in the model: any element field matches with an element wnull. We have been forced to define an extra element wnull because, using XML-schema, it is not possible to define a top type, that is an XML-schema such that any document results to be valid with respect to such schema. An example of template is reported in Table 4.2.

The matching rule of WSSecSpaces differs from the formal definition (see Definition 3.1) in the evaluation of the matching between data fields: they match if each element field of the entry matches with the corresponding element (field, wfield or wnull) of the template. More formally, let WField, ranged over by wf, wf', ..., be the set of the possible contents of wfield elements. We model the content of field elements with the data fields (d) defined in Section 2, whilst now the content of the fields used by templates is defined as follows: dt := d | wf | wnull. Let $\geq_{\text{XML}}$ be a relation between data fields and wildcard fields: we say that $d \geq_{\text{XML}} w$ if $d$ can be validated using the schema $w$.

**Definition 4.1 XML-based matching rule** – Let $e = \langle d_1; d_2; \ldots; d_n \rangle_{[c_1; d_1; c_2; d_2; \ldots; d_n]}$ be an entry, $t = \langle dt_1; dt_2; \ldots; dt_m \rangle_{[c_1]}$ be a template and $\text{op} \in \{\text{rd}, \text{in}\}$ be an operation. We say that $e$ matches $t_{\text{op}}$ if conditions 1, 3 and 4 of Definition 3.1 and the following condition hold:
2'. \( dt_i = \text{wnull} \text{ or } (dt_i \triangleright_{\text{XML}} dt_i \text{ if } dt_i \in \text{WField}) \text{ or } d_i = dt_i, 1 \leq i \leq n \).

It is easy to observe that this rule is a conservative extension of the original matching rule. Furthermore, the access control mechanisms of \textit{WSSecSpaces} have the same properties of \textit{SecSpaces} because they are based on control fields; therefore the matching rule on data fields does not affect these properties (for more details, see chapter 2).

It is worth noting that the matching rule can be implemented using standard tools for XML processing. Moreover, in the case of typed wildcards the termination of the validation process is ensured. This is an important aspect because this ensures that no unexpected exceptions or non–terminating evaluations of the matching rule can be forced by malicious clients of \textit{WSSecSpaces}.

Finally, the schema describes the structure of the return values of the operations (\textit{ret}); it can be composed only of either a sequence of data fields (\textit{datafield}) or of the element \textit{noValue} that denotes the absence of a return value; the former is used for \textit{rd/in} operations whilst the latter for \textit{out} ones.

### 4.2.3 WSDL and implementation

The Web Services Description Language (WSDL) is a XML-based language used to describe and publish information concerning the service, such as the format of the exchanged information and the modalities a client may use to interact with the service.

In this section we comment only the significant design aspects of the WSDL document associated to \textit{WSSecSpaces}, for the remaining details interested readers can find the complete WSDL source of \textit{WSSecSpaces} in A.2. Since all coordination primitives provide a return value the interaction is always of kind \textit{send-response}; therefore we have opted for a solution with only one access port–whose location address exploits the HTTPS protocol– where the operation the client is willing to perform is described in the content of the SOAP documents used for service invocations.
From the point of view of the internal implementation, a Java Servlet, that has been tested by using Tomcat as servlets container, implements the core of WSSEcSpaces. Finally, the implementation exploits JAXP for Java XML processing operations and the standard security Java package for public key encryption algorithm; more precisely, the actual implementation exploits the DSA algorithm.

4.3 Related work

In the years we assisted to a renewed interest on Linda-like coordination languages from both implementation and theoretical point of views. In particular, three main aspects have attracted the attention of the coordination community: i) security in coordination; ii) integration of Linda with Web technologies (e.g., XML); iii) coordination of Web Services. WSSEcSpaces tries to meet all these aspects. The SecSpaces coordination model and its related work has already been compared with other proposals supporting some form of secure coordination in Chapter 2.

Considering the integration of XML in Linda-like platforms, one former proposal is XMLSpaces, a coordination middleware –based on TSpaces– that exploits XML: entries and templates are XML documents and XML-query, as well as DTD (Document Type Definition), are used in the matching rule. For a complete survey of XML-based coordination infrastructures see [CTZ02].

The third category we consider is Linda and Web Services and, to the best of our knowledge, the most interesting proposals are [aBMM+03] and Ruple. The former is an implementation of a Web service –based on JavaSpaces– that allows for a location-based coordination of Web services (i.e. different locations support different coordination spaces). The latter is an implementation of a coordination infrastructure for Web services and it is the only related work that supports both security and XML-technologies; it provides an access control mechanism based on digital certificates that does not discriminate between the destruc-
tive and non-destructive read operations. Further, the use of digital certificates bounds the possible readers/consumers of a datum at creation time; in this way it is not supported a dynamic privileges acquisition on the data that are already available in the repository.

4.4 Conclusion

In this chapter we have discussed the design aspects and the implementation of WSSecSpaces, a data-driven coordination service to be exploited in Web Services applications. As already discussed in the Introduction, loose coupling among Web Services is one of the main advantages introduced by WSSecSpaces. In other terms, WSSecSpaces represents in the setting of Web Services what JavaSpaces [FHA99] is for Jini [Sun99] technology. Moreover, WSSecSpaces supports a more sophisticated pattern matching mechanism that permits to control the access to the data available in the coordination space, as well as the authentication of the producer of a specific datum.

Actually, the security of data transmitted on channel is preserved by HTTPS transfer protocols used in the SOAP service invocation protocol, the study of alternatives based on other technologies is left as future work. For instance, there are some other interesting technologies such as XML-encryption [w3c] that can be coherently combined with the Web Services systems and SecSpaces. Further, we intend to investigate on the definition of a Web Services coordination infrastructure that takes into account quality of service requirements, e.g. based on the configuration/topology of the system.
Chapter 5

Combining partitions in \textit{SecSpaces}

In this chapter we consider one of the limitations of \textit{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 \textit{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.

5.1 Introduction

In this chapter we investigate, in the context of \textit{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, for instance, 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.

The native \textit{SecSpaces} model is not particularly suited to support applications with these 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 mod-
ify 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 performs 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 between the \(p\) and \(p'\) partitions.

In the chapter 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 chapter is structured as follows: in Section 5.2 we extend the process calculus introducing the possibility to create new partitions as well as to combine partitions, in Section 5.3 and 5.4 we describe some applications that demonstrate the flexibility of the new extensions in order to model several access control policies, and in Section 5.5 we report some conclusive remarks.

### 5.2 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 it is possible to perform the search of a matching entry (when reading or consuming). The language we are going to presents extends SecSpaces by introducing the merge operator on partitions and an operator for the generation of new names for partitions. In Section 5.2.1 we present the extended language and the new definition of the matching rule, while Section 5.2.2 introduces the corresponding semantics.

#### 5.2.1 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 the SecSpaces coordination primitives and the state of the shared tuple space.
Let C Partition, 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 \( ps : \text{C Partition} \rightarrow \mathcal{P}(\text{Partition}) \) as the function that, given a combination of partitions, returns the set of partitions it contains, whose definition is the following:
\[
ps(c) = \{c\}, \quad ps(p_1 : p_2) = ps(p_1) \cup ps(p_2).
\]

Entry and template structure change in the symmetric partition fields that now contain, instead of a partition \( c \in \text{Partition} \), a combination of partitions \( p \in \text{C Partition} \). An entry \( e = \langle d_1; d_2; \ldots; d_n \rangle^{[p_1, \ldots, p_{m+1}]} \) 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 \( t = \langle d \rangle_{\text{rd} \text{in}} \) means that the matching entry must be available in at least one partition in \( ps(p) \). The definition of the matching rule between entries and templates follows.

**Definition 5.1 Matching rule with combined partitions** – Let
\[
e = \langle d_1; d_2; \ldots; d_n \rangle^{[p_1, \ldots, p_{m+1}]} \text{ be an entry, } t = \langle d t_1; d t_2; \ldots; d t_m \rangle^{[p_1]} \text{ be a template and } op \in \{\text{rd, in}\} \text{ be an operation. Let } p_c \text{ be the combined partition of } e \text{ associated to } op, \text{ we say that } e \text{ matches}_{\text{op}} t \text{ if conditions (i), (ii) and (iv) of Definition 3.1 and the following condition hold:}
\]
\[
(iii)' \quad ps(p_c) \cap ps(p) \neq \emptyset.
\]

Condition (iii)' (which replaces (iii) of Definition 3.1) 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.

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

---

1We assume that associative property for the merge operator holds, that is \( \{c : c'\} : c'' = c : \{c' : c''\} \).
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 \texttt{rd}_P and \texttt{in}_P primitives corresponding to the \texttt{rd} and \texttt{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 \textit{c} and \textit{c}' (the extension to a generic number is straightforward), they can exploit non-blocking primitives by alternatively performing the access to partition \textit{c} and to \textit{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 \texttt{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 \texttt{Var}, ranged over by \textit{x}, \textit{y}, \ldots, be the set of data variables. In the following, we use \(\bar{x}, \bar{y}, \ldots\), to denote finite sequences \(x_1;x_2;\ldots;x_n\) of data variables.

System configurations, ranged over by \textit{A}, \textit{B}, \ldots, and processes, ranged over by \textit{P}, \textit{Q}, \ldots, are defined as follows:

\[
\begin{align*}
A, B, \ldots := &\quad \text{systems} \\
&\quad \text{entries} \\
P &\quad \text{processes} \\
A | A &\quad \text{parallel composition} \\
(\forall c) A &\quad \text{restriction}
\end{align*}
\]
A system can be an entry, a process or the parallel composition of entries and processes. \((\forall c)\ A\) means that the partition name \(c\) is bound in the system \(A\). A process can be a terminated program \(0\), 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 \(\mu\) can be one of the following classical Linda operations: i) \(\text{out } e\), that writes the entry \(e\) in the TS; ii) \(\text{rd } t[\bar{x}].P\), that given a template \(t\) reads a matching entry \(e\) in the TS and stores the return value in \(\bar{x}\); iii) \(\text{in } t(\bar{x}).P\), that given a template \(t\) consumes a matching entry \(e\) in the TS and stores the return value in \(\bar{x}\). A process \(P | Q\) is the parallel composition of two processes \(P\) and \(Q\) which behaves as the processes \(P\) and \(Q\) running in parallel. Recursive processes 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): \((\forall c)\ P\) 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 \(P[\tilde{d}/\tilde{x}]\) to denote the process obtained by replacing in \(P\) all occurrences of variables in \(\tilde{x}\) with the corresponding value in \(\tilde{d}\), that is \(P[d_1;/d_2;\ldots;d_n/x_1;x_2;\ldots;x_n] = P[d_1/x_1][d_2/x_2]\ldots [d_n/x_n]\).
Chapter 5. Combining partitions in SecSpaces

We say that a system is well formed if each rd/in < d_t \{ c_r, c_l \}_\ln (\bar{x}) operation is such that the variables \bar{x} and the tuple of data d_t 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.

5.2.2 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 5.1 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 5.2 contains the system reduction rules. Rules (1), (2) and (3) describe the three prefix operators in, rd, and out, respectively. More precisely: (1) shows that the process in t(\bar{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[e/\bar{x}]; (2) shows that the process rd t(\bar{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[e/\bar{x}]; and (3) shows that out e.P produces in one step a new occurrence of the entry e into the
Table 5.1: Structural equivalence over systems

| (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) | $A \mid 0 \equiv A$ | (viii) | $!P \equiv P \mid !P$ |
| (ix) | $(\forall c) (\forall c') A \equiv (\forall c') (\forall c) A$ | (x) | $(\forall c) 0 \equiv 0$ |
| (xi) | $(\forall c) (A \mid B) \equiv (\forall c) A \mid B$, if $c \notin \text{fn}(B)$ |
| (xii) | if $A$ can be turn to $B$ by alpha-conversion then $A \equiv B$ |
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 5.1 and rules (1) and (2) allow us to implement the scope extrusion of partition names as originally proposed in the π-calculus [MPW92].

\[
\begin{align*}
(1) & \quad e \text{ matches}_{\text{in}} t \\
& \quad e \mid \text{in } t(\bar{x}).P \rightarrow P[\bar{e}/\bar{x}] \\
(2) & \quad e \text{ matches}_{\text{rd}} t \\
& \quad e \mid \text{rd } t(\bar{x}).P \rightarrow e \mid P[\bar{e}/\bar{x}] \\
(3) & \quad \text{out } e.P \rightarrow e \mid P \\
(4) & \quad A \rightarrow A' \\
& \quad A \mid B \rightarrow A' \mid B \\
(5) & \quad A \equiv B \quad B \rightarrow B' \\
& \quad A \rightarrow B' \\
(6) & \quad A \rightarrow A' \\
& \quad (\forall c)A \rightarrow (\forall c)A'
\end{align*}
\]

**Table 5.2: Transition system**

### 5.3 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 tasks. In previous chapter we have already proved that some security properties in the interaction among processes can be guaranteed. 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 recall the idea to implement a secure group communication, where only entities of the group can access the entries used in the group commu-
nication. 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):

```
(B,4) (A,5) (B,5) (D,6) (C,7) (B,4) (D,2) (B,5) (C,2)
```

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 5.3.1.

While in the previous chapters the secure group communication was described assuming that the entities of the group were statically defined, here we investigate the issues emerging when the group configuration can dynamically change. 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 5.2.2), 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 5.3.1 explains how to manage the group restriction by exploiting the merge operator. Finally, Section 5.3.2 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.
5.3.1 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 $G_p = \{P_{i_1}, P_{i_1+1}, \ldots, P_{i_s}\} \subseteq G$ 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:

$$(\forall c) (P_1 | \ldots | P_{i-1} | (\forall c') (P_{i} | \ldots | P_{i+s} | \ldots | P_n)).$$

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., $\text{out}(\langle d \rangle[c']^{rd}[c']^{ln} ; 2)$ 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 $\text{rd} \langle n u l l \rangle^{[cc']}(x)$. A particular instance of this problem is the removal of some users from the group: by combining the partition $c$ with the new partition $c'$ we can 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 5.1 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 $k_{ij}$ is known only by $P_j$. The group restriction can be implemented by exploiting the following distribution protocol:

$$(\forall c) (P_1 | \ldots | P_{i-1} | P_i' | \ldots | P_s' | \ldots | P_n)$$

where

$$P_i' = (\forall c') \text{ out} (\langle c' \rangle_{[c_r,d,c_l]}_{[k_{i(t+1)},r,d,k_{i(t+1)}]} \text{ out} (\langle c' \rangle_{[c_r,d,c_l]}_{[k_{i(t+2)},r,d,k_{i(t+2)}]} \ldots \text{ out} (\langle c' \rangle_{[c_r,d,c_l]}_{[k_{i(t)},r,d,k_{i(t)}]}), P_i, \text{ and}$$

$$P_j' = \text{ in } \langle \text{null} \rangle_{[c]}_{[k_{j(i)}]} (x) \text{ for } i + 1 \leq j \leq s.$$ 

$P_i'$ creates a new partition $c'$ and then distributes it to the processes in $G_p$, thus reaching the configuration $(\forall c) (P_1 | \ldots | P_{i-1} | (\forall c') (P_i | \ldots | P_s) | \ldots | P_n)$. It should be clear that the entry $\langle c' \rangle_{[c_r,d,c_l]}_{[k_{i(t)},r,d,k_{i(t)}]}$ can be produced only by $P_i'$ and that only $P_i'$ can access that entry, for any $i + 1 \leq j \leq s$.

### 5.3.2 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 that other groups know; 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 it is willing to publish (e.g., $\text{out}(\langle g' g, d \rangle^{c \mid r \mid d \mid c \mid r})$). The coordinator removes such entries, logs the request and then publishes the entry. The coordinator process definition follows:

$$(!\text{in } \langle \text{null; null} \rangle^{c \mid c'}(x_1; x_2).\text{LogReq}(x_1, x_2).\text{out}(\langle x_2 \rangle^{P(x_1)} || r || d || P(x_1) || \text{in})))$$

where $\text{LogReq}$ registers the request and $P(g \downarrow P)$ returns the expression that merges the partitions identified in $g \downarrow P$ (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.

### 5.4 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 its child.

The model we intend to encode in SecSpaces allows a process 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) first express the multilevel system extending SecSpaces introducing a new operator used to generate sub-partitions, and ii) we encode the new sub-partition operator by using the merge operator proposed in the previous section.

The partition field values we use are defined by the following grammar:
\[
p ::= c \mid p \rightarrow c.
\]

Given a partition \( p_1 \), the sub-partition operator allows us to introduce a new partition \( p_1 \rightarrow c \) which is a sub-partition of \( p_1 \). The sub-partition relation is transitive in the sense that \( p \rightarrow c_1 \rightarrow c_2 \) is a sub-partition of \( p \rightarrow c_1 \) as well as of \( p \). In other terms, a partition is a sub-partition of another one if the name of the latter is a prefix of the former one.

The idea we follow is that partitions defined above 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 \( p_1 \rightarrow c \) expresses that \( p_1 \) is an ancestor of \( p_1 \rightarrow c \) and then \( p_1 \) has a level higher than \( p_1 \rightarrow c \). We also define the function \( as \) that, given a partition \( p \), returns the set of nodes which are ancestors of \( p \), whose definition follows:

\[
as(p_1 \rightarrow c) = \{p_1 \rightarrow c\} \cup as(p_1), \quad as(c) = \{c\}.
\]
By using the sub-partition operator to combine partitions, the matching rule is defined by replacing condition (iii)' in the Definition 5.1 with the condition $p_c \in as(p_t)$. In this way, the matching rule allows someone to match entries that appear either in the partition specified by the template or in each partition with an higher level.

In order to express the multilevel system, we assume that the knowledge of the security level identifiers (i.e. partitions which also represent hierarchical level relationship via the $\rightarrow$ 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:

- $out(\vec{d}, p) = out(\langle\vec{d}\rangle^{[p]}_{l, d, l, r})$, that inserts the datum $\vec{d}$ in the level $p$;
- $rd(\vec{d}t, p) = rd(\langle\vec{d}t\rangle^{[p]}_{l, r})$, that reads a datum matching with $\vec{d}t$ by using access level $p$, and
- $in(\vec{d}t, p) = in(\langle\vec{d}t\rangle^{[p]}_{l, r})$, that consumes a datum matching with $\vec{d}t$ 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 $p \rightarrow c$ can read/consume any data available at a level in $as(p \rightarrow c)$.

Finally, here we show how to encode the sub-partition operator by exploiting the language with the merge operator presented in Section 5.2.1. 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 in $as(p)$. For instance, $out(\vec{d}, c_1 \rightarrow c_2)$ is encoded with $out(\langle\vec{d}\rangle^{c_1 \rightarrow c_2}_{l, d, l, r})$ that produces an entry in a partition that is to an higher level than the one used by $rd(\vec{d}t, (c_1 \rightarrow c_2) \rightarrow c_3)$, encoded with the primitive
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.

### 5.5 Conclusion

In this chapter 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, it could be interesting 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 5.4.
Chapter 6

Probabilistic and Prioritized Data Retrieval in the Linda Coordination Model

Linda tuple spaces are flat and unstructured, in the sense that they do not allow for expressing preferences of tuples; for example, we could be interested in indicating tuples that should be returned more frequently w.r.t. other ones, or even tuples with a low relevance that should be taken under consideration only if there is no tuple with a higher importance. In this chapter we investigate, in a process algebraic setting, how probabilities and priorities can be introduced in the Linda coordination model in order to support a more sophisticated data retrieval mechanism. As far as probabilities are concerned, we show that the Linda pattern-matching data retrieval makes it necessary to deal with weights instead of just pure probabilities, as instead can be done in standard process algebras. Regarding priorities, we present two possible ways for adding them to Linda; in the first one the order of priorities is statically fixed, in the second one it is dynamically instantiated when a data-retrieval operation is executed.

6.1 Introduction

The native Linda coordination model is based on a shared tuple-space which is an unstructured and flat bag of tuples. All tuples have the same importance and relevance inside the shared repository, in the sense that when several tuples match
the template of a data-retrieval operation, one of them is selected non deterministically. In some applications, we may be interested in expressing more sophisticated policy for selecting the tuple to be returned, for example, according to some priority based access (one tuple should be returned only if no other tuples of higher priority are currently available) or a probabilistic selection (one tuple should be returned with a higher probability w.r.t. another one).

Consider, for example, the problem of coordinating the collaboration among Web-Services; in particular, consider the problem of discovering a Web-Service willing to offer a particular service. A tuple-space could be exploited in this scenario as a registry where the available Web-Services register the kind of services they intend to offer, while the clients access the tuple-space in order to discover the actual Web-Services availability. In the case more than one service is willing to offer the required service, according to the standard data-retrieval mechanism, one is chosen non deterministically. This is not satisfactory if we intend to distribute in a balanced way the workload, thus to avoid the overwhelming of requests towards one Web-Service while leaving other Services under-utilized.

As a second example, consider a master-worker application where the masters and the workers coordinate via a tuple-space: the masters produce job requests and store them inside the tuple-space, and the workers access the tuple-space to retrieve the description of the jobs to execute. We could assume that the jobs have different urgency levels, and that the workers must select a job for execution only if no jobs are currently registered with a higher priority. Another more sophisticated form of priority could be related to time constraints, e.g. assuming that a job must be executed before an expiration time, otherwise it can be left unexecuted. In this case, in order to limit the number of unexecuted jobs, each worker should select the job with the closest expiration time.

In the traditional Linda model these coordination patterns are rather difficult to be programmed. The reason is that the standard Linda data-retrieval coordination primitives (in and rd) access the tuple space on the basis of a local property, i.e., the conformance of the tuple w.r.t. the indicated template independently of
the context in which the tuple is actually inserted. On the other hand, in the above examples *global* properties involving the entire group of matching tuples come into play. More precisely, in the examples the data-retrieval operation should select the tuple to be returned according to some function (either probabilistic or priority-based) that has all the matching tuples in its domain.

Extensions of Linda exist that support *global* operations. Consider, e.g., the collect primitive of [RW98] (that permits to withdraw all the tuples satisfying the template) or the non blocking \texttt{inp} operation supported in some Linda system [CA95] (that returns one tuple matching the template, if available, or terminates indicating the absence of matching tuples). These primitives permit to program the coordination patterns described above, but in a rather unsatisfactory way.

For example, if we want to force a specific probabilistic distribution of the returned tuple, we could decorate each tuple adding (as an extra field) a value that quantifies the level of relevance of the tuple. When a data-retrieval operation is executed from an agent, this agent could collect all the tuples satisfying the template, select the tuple according to the distribution of these values, and re-introduce the tuples in the space. Clearly, this pattern is not satisfactory because it requires to move from the tuple-space to the agent (and back) possibly huge quantities of tuples, and moreover this complex operation should be executed in a transactional manner, thus requiring consistent locks.

As far as the master-worker example is concerned, the priority-based coordination pattern that it requires could be programmed exploiting the \texttt{inp} operation. The level of priority could be associated to the tuples as an extra field. The workers that access the space could initially perform an \texttt{inp} taking into account the first level of priority, and passing to the subsequent levels only if no tuples are retrieved. This solution is satisfactory only if few levels of priority are considered, because it is necessary to explicitly access one level at a time. If the priorities are expressed in terms of an expiration time as described in the example above, this solution is clearly unfeasible because it requires a separate \texttt{inp} operation for each
possible expiration time.

In light of these observations, we consider worth to investigate extensions of the Linda tuple-space coordination model with probabilistic as well as prioritized data retrieval. By probabilistic access we mean the possibility to specify the probabilistic distribution according to which the data to be returned are selected when a template is matched by more than one tuple. We observe that the most reasonable way for specifying this distribution is to associate weights to tuples indicating the absolute relevance of one tuple; when a data retrieval operation is performed, the weights of all the tuple matching the template are taken into account, and the tuple is probabilistically selected according to its relative weight w.r.t. the weights of the other tuples. Using this approach we can satisfactorily solve the problem of a balanced distribution of the workload of the Web-Services: each Web-Service indicates with a weight its current workload (the higher is the workload, the lower is the weight). When a client performs its discovery operation, a link to a Web-Service currently unloaded is more probably returned w.r.t. an overwhelmed one.

As far as the priority-based access is concerned, we investigate two possible scenarios; in the first one the order of the priorities is statically fixed; in the second one tuples are associated with a symbolic priority and the order relation among the symbolic priorities is defined from the processes at the time they execute the data-retrieval operations. Going back to the master-worker example, the first approach can easily solve the case in which there is a fixed hierarchy of urgency levels, simply by associating a different priority to each urgency level. On the other hand, in the case the job with the closest expiration time should be selected, we can proceed as follows: the expiration time of a job indicates its symbolic priority and, when a worker withdraws a job, it defines an order relation that privileges those with an expiration which is closer to the current time.

The chapter is structured as follows. In Section 6.2 we present a formal description of the Linda coordination model that we use as the basis for our probabilistic and prioritized extensions. In Section 6.3 we discuss and formally intro-
duce our probabilistic version of Linda, while in Section 6.4 we discuss and define the two extensions with a prioritized access to tuples. Finally, in Section 6.5 we draw some conclusive remark.

6.2 The Linda coordination model

The coordination primitives that we have in Linda are: \texttt{out}(e), \texttt{in}(t) and \texttt{rd}(t). The output operation \texttt{out}(e) inserts a tuple \(e\) in the tuple space (TS for short). Primitive \texttt{in}(t) is the blocking input operation: when an occurrence of a tuple \(e\) matching with \(t\) (denoting a template) is found in the TS, it is removed from the TS and the primitive returns the tuple. The read primitive \texttt{rd}(t) is the blocking read operation that, differently from \texttt{in}(t), returns the matching tuple \(e\) without removing it from the TS.

Linda tuples are ordered and finite sequences of typed fields, while template are ordered and finite sequences of fields (see [CA95]) that can be either actual or formal: a field is actual if it specifies a type and a value, whilst it is formal if the type only is given. For the sake of simplicity, in the formalization of Linda we are going to present fields are not typed.

Formally, let \texttt{Mess}, ranged over by \(m, m', \ldots\), be a denumerable set of messages and \texttt{Var}, ranged over by \(x, y, \ldots\), be the set of data variables. In the following, we use \(\bar{x}, \bar{y}, \ldots\), to denote finite sequences \(x_1; x_2; \ldots; x_n\) of variables.

Tuples, denoted by \(e, e', \ldots\), are finite and ordered sequences of data fields, whilst templates, denoted by \(t, t', \ldots\), are finite and ordered sequences of fields that can be either data or wildcards (used to match with any message).

Formally, tuples are defined as follows:

\[
e = \langle \bar{d} \rangle
\]

where \(\bar{d}\) is a term of the following grammar:

\[
\bar{d} ::= d \mid d; \bar{d}
\]
The definition of template follows:

\[ t = \langle \tilde{d}t \rangle \]

where \( \tilde{dt} \) is a term of the following grammar:

\[
\begin{align*}
\tilde{dt} & := dt \mid dt; \tilde{dt} \\
dt & := d \mid \text{null}.
\end{align*}
\]

A data field \( d \) can be a message or a variable. The additional value \( \text{null} \) denotes the wildcard, whose meaning is the same of formal fields of Linda, i.e. it matches with any field value. In the following, the set \( \text{Tuple} \) (resp. \( \text{Template} \)) denotes the set of tuples (resp. templates) containing no variable.

The matching rule between tuples and templates we consider is the classical one of Linda, whose definition is as follows.

**Definition 6.1 Matching rule** - Let \( e = \langle d_1; d_2; \ldots; d_n \rangle \in \text{Tuple} \) be a tuple, \( t = \langle dt_1; dt_2; \ldots; dt_m \rangle \in \text{Template} \) be a template; we say that \( e \) matches \( t \) (denoted by \( e \triangleright t \)) if the following conditions hold:

1. \( m = n \).

2. \( dt_i = d_i \) or \( dt_i = \text{null}, 1 \leq i \leq n \).

Condition 1. checks if \( e \) and \( t \) have the same arity, whilst 2. tests if each non-wildcard field of \( t \) is equal to the corresponding field of \( e \).

Processes, denoted by \( P, Q, \ldots \), are defined as follows:

\[
\begin{align*}
P, Q, \ldots := & \quad \text{processes} \\
0 & \quad \text{null process} \\
\mid \ & \text{out}(e).P \\
\mid \ & \text{rd} t(\bar{x}).P \\
\mid \ & \text{in} t(\bar{x}).P \\
\mid \ & P | P 
\end{align*}
\]

parallel composition
A process can be a terminated program $0$, 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 coordination primitives: i) out $\langle e \rangle$, that writes the tuple $e$ in the TS; ii) $\text{rd } t(\vec{x})$, that given a template $t$ reads a matching tuple $e$ in the TS and stores the return value in $\vec{x}$; iii) $\text{in } t(\vec{x})$, that given a template $t$ consumes a matching tuple $e$ in the TS and stores the return value in $\vec{x}$. In both the $\text{rd } t(\vec{x})$ and $\text{in } t(\vec{x})$ operations $\langle \vec{x} \rangle$ is a binder for the variables in $\vec{x}$. The parallel composition $P \mid Q$ of two processes $P$ and $Q$ behaves as two processes running in parallel, whilst the replication operator $!P$ denotes the parallel composition of infinite copies of $P$.

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

We say that a process is well formed if each prefix operation of kind $\text{rd } / \text{in } \langle \vec{d}t \rangle (\vec{x})$ is such that the variables $\vec{x}$ and the data $\vec{d}$ have the same arity. Notice that in the $\text{rd } t(\vec{x})$ and $\text{in } t(\vec{x})$ operations we use a notation which is different from the standard Linda notation: we explicitly indicate in $\langle \vec{x} \rangle$ the variables that will be bound to the actual fields of the matching tuple, while in the standard Linda notation these variables are part of the template. Observe that the two notations are equivalent up to the fact that our notation introduces variables also in association to the formal fields of the template. We also say that a process is closed if it has no free variable. In the following, we consider only processes that are closed and well formed; Process denotes the set of such processes.

Let $D\text{Space}$, ranged over by $DS, DS', \ldots$, be the set of possible configurations of the TS, that is $D\text{Space} = M_{\text{fin}}(\text{Tuple})$, where $M_{\text{fin}}(S)$ denotes the set of all the possible finite multisets on $S$. In the following, we use $DS(e)$ to denote the number of occurrences of $e$ within $DS \in D\text{Space}$. The set $\text{System} = \{ [P, DS] | P \in
Process, $DS \in DSpace$, ranged over by $s, s', \ldots$, denotes the possible configurations of systems.

The semantics we use to describe processes interacting via Linda primitives is defined in terms of a transition system $(System, \to)$, where $\to \subseteq System \times System$. More precisely, $\to$ is the minimal relation satisfying the axioms and rules of Table 6.1 (symmetric rule of (4) is omitted). $(s, s') \in \to$ (also denoted by $s \to s'$) means that a system $s$ can evolve (performing a single action) in the system $s'$.

$$
\begin{align*}
(1) & \quad [\text{out } (e).P, DS] \to [P, DS \oplus e] \\
(2) & \quad \exists e \in DS : e \triangleright t \\
    & \quad [\text{in } t(\overline{x}).P, DS] \to [P[e/\overline{x}], DS - e] \\
(3) & \quad \exists e \in DS : e \triangleright t \\
    & \quad [\text{rd } t(\overline{x}).P, DS] \to [P[e/\overline{x}], DS] \\
(4) & \quad [P, DS] \to [P', DS'] \\
    & \quad [P \mid Q, DS] \to [P' \mid Q, DS'] \\
(5) & \quad [P, DS] \to [P', DS'] \\
    & \quad ![P, DS] \to [P' \mid !P, DS']
\end{align*}
$$

Table 6.1: Semantics of Linda
Axiom (1) describes the output operation that produces a new occurrence of the tuple \( e \) in the shared space \( DS \) (\( DS \oplus e \) denotes the multiset obtained by \( DS \) increasing by 1 the number of occurrences of \( e \)). Rules (2) and (3) describe the \( \text{in} \) and the \( \text{rd} \) operations, respectively: if a matching \( e \) tuple is currently available in the space, it is returned at the process invoking the operation and, in the case of \( \text{in} \), it is also removed from the space (\( DS - e \) denotes the removal of an occurrence of \( e \) from the multiset \( DS \)). Rule (4) represents a local computation of processes, whilst (5) the replication operator that produces a new instance of the process and copies itself.

6.3 Adding probabilistic data-retrieval to Linda

In this section we extend the Linda language by introducing probabilities for retrieval of tuples stored in the TS. We start by discussing the probabilistic model that we will adopt (Section 6.3.1) and then we present the new extended Linda primitives: their syntax (Section 6.3.2) and their semantics (Section 6.3.3).

6.3.1 Probabilistic model

In Linda expressing a probabilistic choice among entities reacting to a given communication request (e.g. tuples matching a “rd” or “in” request) requires a much more complex mechanism w.r.t. languages employing channel-based communication (like, e.g., those representable by standard process algebras). This is due to the greater complexity of Linda matching-based communication mechanism w.r.t. such a simpler form of communication. If we had channel-based communication then we could just consider probability distributions (i.e. functions assigning probabilities that sum up to 1 to elements of a given domain) over the messages \( a(d) \) actually available on the channel type \( a \); when a receive operation is performed on the channel of type \( a \), the channel would “react” to the request by simply choosing a message \( a(d) \) for some \( d \) according to such a probability
distribution (this is what would happen by adopting either the “reactive model” of probability [vGSS95] or the “simple model” of [Seg95]). When we consider the Linda matching-based communication mechanism, we lose the separation above between the channel type (which decides the set of entities involved in the communication) and the datum d that is read. Since the set of matching tuples \{\vec{d}\} is now established from a template t on data that is chosen by the “rd” or “in” operation, it is unavoidable to deal with the situation in which the set of matching tuples is a proper subset of the domain of a probability distribution: in this case a re-normalization of “probability” values must be done in order to have them summing up to 1 (this is the same situation that arises for the restriction operator in generative models [vGSS95]). Note that the only way to avoid this would be to have an individual probability distribution for each datum \{\vec{d}\} that is present in the shared space (over the several instances of such datum), i.e. by treating each different datum \{\vec{d}\} in the same way as a channel type α in channel-based communication. However, since in this case the channel type would coincide with the datum that is read from the tuple space, reading (or consuming) different tuples having the same channel type (i.e. different instances of the same datum \{\vec{d}\}) would have the same observable effect on the system, hence probability distributions would be useless.

As a consequence of this remark, since when the shared space is accessed the probabilities on matching tuples must be determined by using re-normalization (on the basis of the “selecting power” of the template in the “rd” or “in” operation), it is natural to express probability statically associated to tuples in the space by means of weights [Tof94]. A weight is a positive real number which is associated to an entity that can be involved in a probabilistic choice: the actual probability that the entity assumes in the choice is determined from the associated weight depending on the context, i.e. from the weights of the other entities involved in the choice, by re-normalization.

Example 6.1 We indicate the weight w of a tuple associating the notation \{w\} to the tuple itself. Let us suppose that the tuple space contains three tuples
\((m_1, m_2)[w], (m_1, m_2')[w']\) and \((m_1', m_2'')[w'']\), then the following happens. If the operation \(\text{rd } (\text{null, null})(x_1, x_2)\).P is performed, the variables \(x_1, x_2\) are bound: to \(m_1, m_2\) with probability \(w/(w + w' + w'')\), to \(m_1, m_2'\) with probability \(w'/(w + w' + w'')\), and to \(m_1', m_2''\) with probability \(w''/(w + w' + w'')\). If the operation \(\text{rd } (m_1, \text{null})(x_1, x_2)\).P is performed, the variables \(x_1, x_2\) are bound: to \(m_1, m_2\) with probability \(w/(w + w')\) and to \(m_1, m_2'\) with probability \(w'/(w + w')\). If the operation \(\text{rd } (m_1, m_2)(x_1, x_2)\).P is performed, the variables \(x_1, x_2\) are bound: to \(m_1, m_2\) with probability \(w/w = 1\).

Moreover note that since the structure of the shared space is highly dynamic and tuples are introduced individually in the space, expressing weights associated to tuples seems to be preferable w.r.t. expressing a single probabilistic distribution over all tuples (generative approach of [vGSS95]) which is to be updated by re-normalization to value 1 every time a tuple is added or removed. Therefore, due to the inherent re-normalization behavior of Linda, and to the observations we have made, we adopt, like in [BB00, Bra02], the approach above based on weights.

### 6.3.2 Syntax

Formally, let \(\text{Weight}\), ranged over by \(w, w', \ldots\), be the set of weights, i.e. positive (non-zero) real numbers. Tuples are now decorated with an attribute indicating the associated weight (representing their “appealing degree”), whilst templates have the classical structure as in Linda.

Tuples, denoted by \(e, e', \ldots\), are now defined as follows:

\[
e = \langle \tilde{d} \rangle [w]
\]

where \(w \in \text{Weight}\) and \(\tilde{d}\) is a sequence of data fields (see Section 6.2) \(d\) that are defined by the following grammar:

\[
d ::= m \mid w \mid x.
\]

A data field \(d\) can be now a message, a weight, or a variable. We also define \(\tilde{\cdot}\)
as the function that, given a tuple \( e \), returns its sequence of data fields (e.g. if \( e = \langle \tilde{d} \rangle \ | w \rangle \) then \( \tilde{e} = \tilde{d} \)), and a function \( W \) that, given a tuple, returns its weight (e.g., \( W(\langle \tilde{d} \rangle \ | w \rangle) = w \)).

Similarly to Section 6.2, in the following we make use of Tuple and Template to denote sets of tuples and template. In general here and in the following sections we will assume them to take into account the current definition of tuples and templates. The same holds for the matching rule between tuples and templates, denoted by “\( \triangleright \)”, which is the classical one of Linda defined in Section 6.2 (the matching evaluation does not take into account the weight attributes associated to tuples).

The extended version of the Linda primitives has the following meaning:

- **out\( (e) \)**, where \( e \in \text{Tuple} \) is the output operation; given a tuple \( e \), where \( e = \langle \tilde{d} \rangle \ | w \rangle \), it writes it into the DS.

- **in\( (t) \)**, where \( t \in \text{Template} \); if some tuple \( e \) matching with template \( t \) is available in the DS, the execution of in causes the removal of one of such tuples \( e \) from the space and returns \( \tilde{e} \). The probability of removing a particular tuple \( e = \langle \tilde{d} \rangle \ | w \rangle \in \text{DS} \) with \( e \) that matches \( t \) is the ratio of \( w \) to the sum of the weights \( w' \) in the tuples \( e' = \langle \tilde{d}' \rangle \ | w' \rangle \) in the DS such that \( e' \) matches with \( t \) (taking into account multiple occurrences of tuples).

- **rd\( (t) \)**, where \( t \in \text{Template} \); if some tuple \( e \) matching with template \( t \) is available in the DS, one of such tuples is read and the returned value is \( \tilde{e} \). The probability of reading a particular tuple \( e \) with \( e \) that matches \( t \) is evaluated as in the input case.

Note that this means that the probability of reading a particular matching sequence of data fields \( \langle \tilde{d} \rangle \) contained in the DS is the ratio of the sum of weights \( w \) associated with the several instances of \( \langle \tilde{d} \rangle \) contained in the DS, to the sum of the weights of the tuples \( e' \) in the DS matching with \( t \).

It is worth noting that the probabilistic access to the tuples is at the level of the subspace that the agent can access using a specific template. More precisely, the
probability distribution depends on the weights of all matching tuples stored in the DS.

### 6.3.3 Semantics

In this section we introduce the semantics of systems interacting via the probabilistic model of Linda. In particular, we describe how we replace -in the data retrieval operations- the standard non-deterministic choice of a tuple among the matching ones in the TS, with a probabilistic choice exploiting weights.

Let \( \text{Prob} = \{ \rho : \text{System} \rightarrow [0, 1] \wedge \text{supp}(\rho) \text{ is finite} \wedge \sum_{\text{System}} \rho(s) = 1 \} \), where \( \text{supp}(\rho) = \{ s \mid \rho(s) > 0 \} \), be the set of probability distributions on configurations. The semantics we use to describe systems is defined in terms of probabilistic transition systems \( (\text{System}, \text{Prob}, \rightarrow) \), where \( \rightarrow \subseteq \text{System} \times \text{Prob} \). More precisely, \( \rightarrow \) is the minimal relation satisfying the axioms and rules of Table 6.2. \((s, \rho) \rightarrow (s', \rho)\) (also denoted by \( s \rightarrow \rho \)) means that a system \( s \) can reach a generic configuration \( s' \) with a probability equal to \( \rho(s') \). Note that, several probability distributions may be performable from the same state \( s \), i.e. it may be that \( s \rightarrow \rho \) for several different \( \rho \). This means that (like in the simple model of [Seg95]) whenever the system is in state \( s \), first a non-deterministic choice is performed which decides which of the several probability distributions \( \rho \) must be considered, then the next configuration is probabilistically determined by the chosen distribution \( \rho \). Note that the non-deterministic choice may, e.g., arise from several concurrent rd operations which probabilistically retrieve data from the tuple-space. We use \( s \rightarrow s' \) to denote \( s \rightarrow \rho \), with \( \rho \) the trivial distribution which gives probability 1 to \( s' \) and probability 0 to all other configurations.

Table 6.3 defines: (i) the probability distributions \( \rho^p_{\text{in } t[X], \text{P } \text{DS}} \) and \( \rho^p_{\text{rd } t[X], \text{P } \text{DS}} \) used for in and rd operations, respectively; (ii) the operator \( \rho | Q \) that, given \( \rho \), computes a new probability distribution that accounts for composition with “\( Q \)”. It is worth noting that \( \rho^p_{\text{in } t[X], \text{P } \text{DS}} \) and \( \rho^p_{\text{rd } t[X], \text{P } \text{DS}} \) are defined only for \( t \in \text{Template} \) and \( \text{DS} \in \text{DSpace} \) such that there exists \( e \in \text{DS} : e \triangleright t \) (that is the condition reported in axioms (2) and (3)).
Table 6.2: Semantics of Linda with probabilistic access to tuples
\[ \rho_{\text{un},t}(x), P, DS(s) = \begin{cases} \frac{W(e) \cdot DS(e)}{\sum_{e' \in DS:e' > t} W(e') \cdot DS(e')} & \text{if } s = [P[e/x], DS - e] \\
0 & \text{o.w.} \end{cases} \]

\[ \rho_{\text{rd},t}(x), P, DS(s) = \begin{cases} \frac{\sum_{e' \in DS:e' > t \land P[e'/x] = P[e/x]} W(e') \cdot DS(e')}{\sum_{e' \in DS:e' > t} W(e') \cdot DS(e')} & \text{if } s = [P[e/x], DS] \\
0 & \text{o.w.} \end{cases} \]

\[ \rho([P', DS]) \quad \text{if } s = [P' | Q, DS], \]

\[ \rho(Q(s) = \begin{cases} \rho([P', DS]) & \text{if } s = [P' | Q, DS], \\
0 & \text{o.w.} \end{cases} \]

\textbf{Table 6.3:} Probability distributions
Axiom (1) describes the output of the tuple e, after the execution an occurrence of e is added to the shared space DS and the process continues with P. Axiom (2) describes the behaviour of in operations; if a tuple e matching with template t is available in the DS, the in execution produces the removal from the space of e and then the process behaves as P[e/x]. The probability of reaching a configuration where a matching tuple e contained in the DS is removed is the ratio of the total weight of the several instances of e in the DS, to the sum of the total weights of the several instances of the matching tuples currently available in the DS. In this way, the probability to reach a system configuration takes into account the multiple ways of removing e due to the several occurrences of e in the DS. The axiom (3) describes rd operations; if a tuple e matching with template t is available in the DS, then the process behaves as P[e/x]. Differently from the previous axiom, rd operations do not modify the tuple space, i.e. reached states do not change the configuration of DS, therefore they are simply differentiated by the continuation P[e/x] of the reading process. For example, let us consider two different tuples e' and e'' that contain the same value in some of their fields. If the reading process considers only the common fields, we have that P[e'/x] = P[e''/x], thus it is not possible to discriminate the selection of the two different tuples. Therefore, the probability of reaching a configuration s that is obtained by reading a tuple e matching with t in the DS (yielding value e) is the ratio of the sum of the total weights associated to the several instances of tuples e' matching with t in the DS such that the continuation of the reading process obtained by reading tuple e' is the same as the one obtained by reading e, to the sum of the total weights of the several instances of the matching tuples currently available in the DS. Rule (4) describes the behaviour of the parallel composition of processes (the symmetric rule is omitted): if configurations reachable from [P, DS] are described by the probability distribution ρ, and P performs an action in the system [P | Q, DS] (the process that proceeds between P and Q is non-deterministically selected), then the reachable configurations are of the form [P' | Q, DS'], for some P' ∈ Process and DS' ∈ DSpace. The probability values of such configurations do not depend
on Q (that is “inactive”) and are equal to $\rho([P', DS'])$. Finally, rule (5) describes the behaviour of process replication operator: $!P$ behaves as an unbounded parallel composition of the process $P$.

6.4 Adding prioritized data-retrieval to Linda

In this section we introduce a manner for expressing priority on the tuples stored inside the TS. Informally, priorities on tuples represent an absolute preference of the currently available tuples in the shared space. More precisely, if a process performing a rd/in operation receives as the return value a tuple $e$, there is no currently available matching tuple $e'$ in TS such that its priority level is greater than the one of $e$. In Section 6.4.1 we present a model with *static priorities*: priorities on tuples are set by the output operations and the data-retrieval operations return a matching tuple with the highest priority among the available ones. In Section 6.4.2 we propose another solution which supports *dynamic priorities*: tuples are partitioned by associating a partition name with each tuple (set by the output operations) and processes performing rd/in operations can express a priority level to be dynamically assigned to partitions for the data-retrieval.

6.4.1 Static priorities

Following the approach used to introduce weights, tuples are now further extended by adding, similarly as in [BB00, Bra02], an additional attribute indicating their priority. Templates keep the classical structure as well as the classical Linda matching rule given in Definition 6.1.

Formally, let $\text{Priority}$, ranged over by $l, l', \ldots$, be the set of possible priorities, i.e. positive (non-zero) natural numbers.

Tuples are now defined as follows:

$$e = \langle \tilde{a} \rangle [w, l]$$
where \( w \in \text{Weight}, l \in \text{Priority} \) and \( \tilde{d} \) is a sequence of data fields \( d \) that are defined by the following grammar:

\[
d := m \mid w \mid l \mid x.
\]

A data field \( d \) now can be a message, a weight, a priority level, or a variable.

In the following, we denote with \( \Pi_l \) the function that, given a tuple, returns its priority level (e.g., if \( e = \langle \tilde{d} \rangle [w, l] \) then \( \Pi_l(e) = l \)), and with \( \mathcal{D}_l \) the partition of \( \mathcal{D} \) determined by selecting all the tuples \( e \) with priority level \( l \), i.e. the multiset contained in \( \mathcal{D} \) such that for any \( e \in \text{Tuple} \) if \( e \in \mathcal{D}_l \) then \( \Pi_l(e) = \Pi_l(e) = l \), otherwise \( \Pi_l(e) = 0 \). We also define \( L(\mathcal{D}, t) \) as the function that, given a space \( \mathcal{D} \) and a template \( t \), returns the highest priority level of tuples matching with \( t \), i.e. \( L(\mathcal{D}, t) = \max\{l \mid \exists e \in \mathcal{D}_l : e \triangleright t\} \). Note that the function \( L(\mathcal{D}, t) \) is defined only in the case \( \mathcal{D} \) contains at least one matching tuple within \( \mathcal{D} \) and, since we consider only \( \mathcal{D} \) that are finite multisets of \( \text{Tuple} \), it can be computed for any \( \mathcal{D} \in \text{DSpace} \).

The semantics of the Linda model with priority and weights on tuples is obtained from Table 6.2, by replacing \( p^{\text{p}}_{\text{in}(t,\mathcal{X}) \cdot P \cdot \mathcal{D}} \) and \( p^{\text{p}}_{\text{in}(t,\mathcal{X}) \cdot P \cdot \mathcal{D}} \) with \( p^{\text{p,1}}_{\text{in}(t,\mathcal{X}) \cdot P \cdot \mathcal{D}} \) and \( p^{\text{p,1}}_{\text{in}(t,\mathcal{X}) \cdot P \cdot \mathcal{D}} \), respectively, which are defined in Table 6.4. The definition of the new probability distributions in Table 6.4 makes use of the probability distributions previously defined in Table 6.3 for the pure probabilistic case. Informally, the idea is that the search space is restricted to the partition of \( \mathcal{D} \) which has the greatest priority level and contains a matching tuple. On this subspace, the probabilistic data-retrieval is governed by weights of the matching tuples it contains. More precisely, \( p^{\text{p,1}}_{\text{in}(t,\mathcal{X}) \cdot P \cdot \mathcal{D}} ([P', \mathcal{D}']) \) is defined by using the corresponding probabilistic version applied on the restricted space, i.e. \( p^{\text{p}}_{\text{in}(t,\mathcal{X}) \cdot P \cdot \mathcal{D} \text{L}(\mathcal{D}, t)} ([P', \mathcal{D}'] \text{L}(\mathcal{D}, t)) \).

This “reduction” is applied only if \( \mathcal{D} \text{L}(\mathcal{D}, t) = \mathcal{D}' \text{L}(\mathcal{D}, t) \) (where “−” is the usual multiset difference operator), that is if the partitions of the space not involved in the restrictions of \( \mathcal{D} \) and \( \mathcal{D}' \) used by \( p^{\text{p}} \) do not change. The same kind of “reduction” and condition have been used to define the probability distribution for \( \text{rd} \) operations.
Table 6.4: Probability distributions with priority


### 6.4.2 Dynamic name-based priorities

Priority attributes on tuples introduced in the previous section logically partition the tuple space: each partition is identified by a priority level and contains all tuples with that specific priority level, i.e. \( DS = \sum_{l \in \text{Priority}} DS_l \), for any \( DS \in \text{DSpace} \).

In this section we discuss how to manage dynamically priorities by providing, at the process that is willing to perform a \( \text{rd}/\text{in} \) operation, a manner to express in which partition of the tuple space to search the matching tuple (by associating an absolute preference to partitions), thus providing a way to restrict the search space to a subset of the tuple space.

Tuples have now an attribute weight and another one that is a “key”: the name used to identify the partition. In order to express in which partitions to search matching tuples and with which priority, templates are decorated by adding a partial function that maps keys on priority levels. In this way, producers of tuples do not assign an absolute preference on the tuples, that is dynamically described by the processes performing a \( \text{in}/\text{rd} \) operation.

Formally, let \( \text{Key} \), ranged over by \( k, k', \ldots \), be the set of keys, and \( \text{KL} = \{ f | f : \text{Key} \rightarrow \text{Priority} \} \), ranged over by \( f, f', \ldots \), be the set of partial functions mapping keys to priority levels; we denote with \( \text{Dom}(f) \) the domain of the function \( f \).

The definition of tuple is as follows:

\[
\epsilon = \langle \dd \rangle [w, k]
\]

where \( w \in \text{Weight} \), \( k \in \text{Key} \) and \( \dd \) is a sequence of data fields \( d \) that can be set to a message, a key, a weight, or a variable. In the following, we use \( \text{PK}(\epsilon) \) to denote the key associated to the tuple \( \epsilon \), i.e. \( \text{PK}(\langle \dd \rangle [w, k]) = k \).

The structure of templates is the following:

\[
t = \langle \ddt \rangle [f]
\]

where \( f \in \text{KL} \) and \( \ddt \) is a sequence of data fields that, in addition to those ones used by tuples, can also be set to wildcard value.
We also define $DS_{[k_1, k_2, \ldots, k_n]}$ as the function that given a $DS \in DSpace$ and a set of keys $\{k_1, k_2, \ldots, k_n | k_i \in \text{Key}, 1 \leq i \leq n\}$ returns the multiset containing all the tuples in $DS$ associated with one of the keys in the given set, i.e. for any $e \in \text{Tuple}$ if $e \in DS$ and $PK(e) = k_i$ for some $1 \leq i \leq n$ then $DS_{[k_1, k_2, \ldots, k_n]}(e) = DS(e)$, otherwise $DS_{[k_1, k_2, \ldots, k_n]}(e) = 0$.

Finally, we define the functions $L(DS, t, f)$ and $K(DS, t, f)$ that, given a configuration of the tuple space $DS \in DSpace$, a template $t \in \text{Template}$ and a function $f \in KL$, return the highest priority level and the set of keys with the highest priority level of the available matching tuples in $DS$, respectively. They are defined as it follows:

$L(DS, t, f) = \max\{f(k) | k \in \text{Dom}(f) \land \exists e \in DS_{[k]} : e \triangleright t\}$

$K(DS, t, f) = \{k | k \in \text{Dom}(f) \land f(k) = L(DS, t, f)\}$

Note that the functions $L$ and $K$ are defined only for $DS$, $t$ and $f$ such that there exists at least one matching tuple $e \in DS$ having an associated key $k$ contained in the domain of $f$.

The semantics of the model with dynamic priority and weights on tuples is obtained by using rules (1), (4) and (5) of Table 6.2, rules (2') and (3') of Table 6.5 and the probability distributions reported in Table 6.6.

Differently from $rd$ and in operations with priority proposed in the previous section, the corresponding ones with dynamic priority can be performed only if a tuple with one of the specified priority levels is found in the space. Therefore, (2') and (3') differ from (2) and (3) of Table 6.2 because we need to test not only the presence in $DS$ of a matching tuple, but also that it has an associated symbol $k$ that is in the domain of $f$ (indicated by the template). The definition of the probability distributions of Table 6.6 follows the same idea used in those ones for static priority: we exploit the definition of probability distributions $\rho^p$ by restricting the space $DS$ to that one containing only tuples having a key corresponding to the highest priority level containing matching tuples.
Table 6.5: rd and in semantics with dynamic priority

\[
\begin{align*}
(2') & \quad \exists e \in DS: e \triangleright t \quad PK(e) \in \text{Dom}(f) \\
& \quad [\text{in} \ (\bar{d}t)[f](\bar{x}).P, DS] \rightarrow \rho_{\text{in}}^{p,dt}(\bar{d}t)[f](\bar{x}).P, DS
\end{align*}
\]

\[
\begin{align*}
(3') & \quad \exists e \in DS: e \triangleright t \quad PK(e) \in \text{Dom}(f) \\
& \quad [\text{rd} \ (\bar{d}t)[f](\bar{x}).P, DS] \rightarrow \rho_{\text{rd}}^{p,dt}(\bar{d}t)[f](\bar{x}).P, DS
\end{align*}
\]

\[
\begin{align*}
\rho_{\text{in}}^{p,dt}(\bar{d}t)[f](\bar{x}).P, DS\left([P', DS']\right) &= \\
& \begin{cases} \\
\rho_{\text{in}}^{p}(\bar{d}t)[f](\bar{x}).P, DS_{k[DS,t,f]}\left([P', DS'_k[DS,t,f]]\right) & \text{if } DS - DS_{k[DS,t,f]} = DS' - DS'_k[DS,t,f] \\
0 & \text{o.w.} \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\rho_{\text{rd}}^{p,dt}(\bar{d}t)[f](\bar{x}).P, DS\left([P', DS']\right) &= \\
& \begin{cases} \\
\rho_{\text{rd}}^{p}(\bar{d}t)[f](\bar{x}).P, DS_{k[DS,t,f]}\left([P', DS'_k[DS,t,f]]\right) & \text{if } DS - DS_{k[DS,t,f]} = DS' - DS'_k[DS,t,f] \\
0 & \text{o.w.} \\
\end{cases}
\end{align*}
\]

Table 6.6: Probability distributions for dynamic priorities
6.5 Conclusion and related work

In this chapter we have introduced, in a process algebraic setting, probabilities and priorities on the data-retrieval mechanisms of Linda. We have technically motivated the use of weights on tuples—instead of classic probability distribution—to express probabilities, and discussed two possible forms of priority mechanisms on tuples, one using static priorities and another one using dynamic priorities defined by the process when they execute data-retrieval operations. To the best of our knowledge, the unique chapter that addresses probabilities in the context of Linda-like languages is Probabilistic Klaim [PHW04]. Klaim [NFP98] is a distributed mobile version of Linda where processes may migrate among nodes in a net; in Probabilistic Klaim probabilities are used to describe probabilistic forms of mobility.
Chapter 7

Priorities and Probabilities in the 
SecSpaces model

In the previous chapter priorities and probabilities in the data-retrieval mechanisms of Linda were introduced. In this chapter we extend the setting of the SecSpaces model in order to include probabilities and priorities, thus supporting all the mechanisms introduced in this thesis. Notably, priorities, either in the static or dynamic form, can be implemented by exploiting the partition fields of SecSpaces.

7.1 Introduction

In this chapter we investigate the possible advantages that can be obtained by combining security mechanisms of SecSpaces with the prioritized and the probabilistic data-retrieval mechanisms.

A natural extension of the Linda model with priorities of the previous chapter, either in the static or dynamic modality, is to provide a manner for limiting the scope of certain levels of priority; in other words, we want to ensure that the tuples associated to a specific priority can be accessed only from a restricted group of processes. Linda is not expressive enough to provide security solutions and, more precisely, access control mechanisms on the tuple space. Therefore, we need
to model priorities on a more sophisticated Linda-like language supporting these features (e.g., SecSpaces, [VBO03, NFP98, MMU01]).

In this chapter we intend to investigate a possible solution that exploits the access control mechanism provided by SecSpaces because it is mainly based on partitions which share common features with the notion of priority. More precisely, each tuple is associated to a partition field that specifies the partition of the tuple space that contains it; processes, in order to perform rd/in operations, must indicate in which partition to perform the search of a matching tuple. In this way, it is possible to limit the access to the tuples stored inside a partition simply by limiting the knowledge of the partition identifier. By exploiting partition fields, we can encode priorities in SecSpaces: each partition represents a priority class and the access on that partition is limited to those processes that know the corresponding partition identifier. We discuss, as a significant case, the implementation of dynamic priorities in the SecSpaces model by exploiting a simple extension of the model proposed in Chapter 5.

We also describe how to extend the SecSpaces model in order to support the probabilistic access to the entries. As an example of application, the probabilistic SecSpaces will be exploited in order to implement a repository intended to be used to distribute the workload among collaborating Web services. More precisely, we describe how to implement a registry service that supports registration, deregistration, discovery and update of Web services descriptions.

The outline of the chapter is as follows. In Section 7.2 we describe how to implement dynamic priorities on tuples in the SecSpaces model. In Section 7.3 we formalize the SecSpaces model with probabilistic data-retrieval and discuss how to extend the WSSecSpaces implementation. In Section 7.4 we prove that the combination of the access control mechanism provided by SecSpaces with the probabilistic access control mechanism, permits the definition of a dynamic Web services discovery protocol that supports controlled as well as balanced Web service interaction. The Web service registry and the discovery mechanism are reported in Section 7.5. Finally, Section 7.6 draws some conclusive remarks.
Chapter 7. Priorities and Probabilities in SecSpaces

7.2 Priorities in the SecSpaces model

In this section we describe how to model dynamic name-based priorities in the SecSpaces model. As mentioned in the Introduction priorities share some features with partition fields because both ones induce a partitioning of the shared space. In order to support priorities, we proceed as follows: i) we consider a direct mapping between symbolic priorities and partition fields, and ii) we provide new constructs for expressing a preference among partitions.

The function $\mathcal{f}$ defined in Section 6.4.2 essentially maps symbolic priorities on natural numbers, that denote the priority levels. It is used to allow processes, that are willing to perform a data-retrieval, to express a preference on entries, that is the priority levels associated to symbolic priorities. Thus, the operators on partitions we need to define should provide: i) a method to aggregate symbolic priorities at a certain priority level, and ii) a method to express the order relation among symbolic priorities. Point i) will be supported by using the merge operator on partitions introduced in Chapter 5, while for the latter we consider a simple sequence operator on structured partitions, denoted by "", in which each partition has a priority level higher than those ones that follow in the sequence.

Formally, let $\text{PPartition}$, ranged over by $p, p', \ldots$, be the set of expressions describing priorities among partitions, whose definition follows:

$$p ::= c \mid p : p \mid p, p$$

where $c \in \text{Partition}$. Similarly to Chapter 5, we define the function $\text{ps} : \text{PPartition} \rightarrow \text{Partition}$ as it follows:

$$\text{ps}(c) = \{c\} \quad \text{ps}(p : p') = \text{ps}(p, p') = \text{ps}(p) \cup \text{ps}(p').$$

The merge operator $p : p'$ expresses that partitions contained in $\text{ps}(p)$ and $\text{ps}(p')$ have the same priority level, while the sequence operator $p, p'$ expresses that partitions in $\text{ps}(p)$ have a priority level higher than those ones contained in $\text{ps}(p')$. For example, $c : c', c_1, c_3 : c_9$ means that $c$ and $c'$ have the highest priority level,
then follows \( c_1 \) and finally \( c_3 \) and \( c_9 \) with the lowest priority. It should be clear that \( p \in \text{Partition} \) can encode any priority function \( f \) defined in Section 6.4.2.

Templates are defined by replacing the content of partition field that now contains a \( p \in \text{Partition} \) instead of \( c \in \text{Partition} \), while entries keep the standard structure. The definition of the matching rule based on the current structure of templates follows.

**Definition 7.1 Matching rule** – Let \( e = \langle d_1; d_2; \ldots; d_m \rangle^{[c_1; c_2; \ldots; c_n]} \) be an entry, \( t = \langle dt_1; dt_2; \ldots; dt_m \rangle^{[p_1; p_2]} \) be a template and \( \text{op} \in \{\text{rd}, \text{in}\} \) be an operation. Let \( c_e \) be the partition of \( e \) associated to \( \text{op} \), we say that \( e \triangleright_{\text{op}} t \) if conditions (i), (ii) and (iv) of Definition 3.1 and the following condition hold:

\[(iii)' \quad c \in ps(p_t).\]

The condition on partition fields tests if the partition of the entry (associated to the operation the agent is performing) is contained in the set of partitions described by the expression. It is care of the data-retrieval mechanism to allow the access only to the partitions with highest priority. The semantics of SecSpaces with priorities can be obtained by replacing rules (2) and (3) of Table 6.1 with those reported in Table 7.1 (\( pf(e) \) is the function that returns the partition field value of \( e \) associated to the data-retrieval operation the agent is performing). Both rules essentially express that the matching entry that is retrieved must have the highest priority level among the matching entries.

In order to produce an entry with a certain symbolic priority, mapped on partition \( c \in \text{Partition} \), processes must set the partition field of the entry to \( c \), while to express the preference among entries the data-retrieval primitives can use the expressions in \( \text{PPartition} \). Notably, in SecSpaces two kinds of control fields come in pair: the control fields to be taken into account when a \( \text{rd} \) is executed and those to be considered in case of \( \text{in} \). For this reason, SecSpaces allows us to refine priorities on tuples at the level of \( \text{rd} \) and \( \text{in} \) operations.

Finally, the prioritized SecSpaces allows us to restrict the access to some symbolic priorities, independently of their priority levels, simply by limiting the
knowledge of partition fields associated with them. This proposal could be used to extend the access control mechanisms of the multilevel system proposed in Section 5.4 by associating, at each level of the system, a priority (that should properly reflect the order relation among the levels of the system). In the solution proposed in that section processes can access any matching entry available at the levels higher than (or equal to) the specified one, while in this case we make it possible to access only to the entries available at the highest priority level among those ones that match the template.

From the point of view of the implementation, the encoding of priorities we propose does not introduce a significant overhead in the pattern matching mechanism. Indeed, as the standard data-retrieval primitives the corresponding prioritized ones can be performed only when a matching entry is available in the TS; in addition, the returned entry must be in the greatest priority level containing a matching entry. This additional condition can be satisfied simply by starting the search of the matching entry from the highest priority level to the lowest one.
7.3 Probabilities in the SecSpaces model

In this section we discuss an extension of the SecSpaces model obtained by adding probabilistic access to the entries stored in the shared space. Technically, we follow the idea of Section 6.3 in which the probabilistic data-retrieval mechanisms of Linda have been presented: we insert an additional attribute to the entries, passed as a parameter to the output operations, that represents the weight of the written entry.

Informally, the weight associated to each entry in the TS represents its appealing degree: among the entries in the TS that can be read/removed by an agent, the entry with greatest weight has the highest probability to be read/removed by the agent. Since the SecSpaces model accepts multiple instances of the same entry, the probability to access a specific entry depends also on the weights associated with the several instances of each matching entry.

Entries stored in the TS are now represented by pairs of the form \((e, w)\), that represent an entry \(e\) having weight \(w\). Let \(\text{Weight}\), ranged over by \(w, w', \ldots\), be the set of weights, i.e. positive (non-zero) real numbers. The extended version of the SecSpaces primitives has the following syntax and meaning:

- \(\text{out} \ (e, w)\), where \(e \in \text{Entry}\) and \(w \in \text{Weight}\) is the output operation; given as parameters an entry \(e\) and a weight \(w\) it writes into the TS a pair \((e, w)\).

- \(\text{in} \ (t)\), where \(t \in \text{Template}\) is the (unchanged) syntax of input operations; if an entry \(e\) matching the template \(t\) is available in the TS, i.e. \((e, w) \in \text{TS}\) for some \(w \in \text{Weight}\), the execution of \(\text{in}\) causes the removal of a pair \((e, w)\) from the space and returns \(\tilde{e}\). The probability of removing a particular pair \((e, w) \in \text{TS}\) with \(e\) that matches \(t\) is the ratio of \(w\) to the sum of the weights \(w'\) in the pairs \((e', w')\) in the TS such that \(e'\) matches with \(t\) (taking into account multiple occurrences of pairs).

- \(\text{rd} \ (t)\), where \(t \in \text{Template}\) is the (unchanged) syntax of read operation; if an entry \(e\) matching the template \(t\) is available in the TS, then the read is
performed and the returned value is $\tilde{e}$. The probability of reading a particular pair $(e, w) \in TS$ with $e$ that matches $t$ is evaluated as in the input case. Note that, in particular, this means that the probability of reading a particular matching entry $e$ contained in the TS is the ratio of the sum of weights $w$ associated with the several instances of the entry $e$ contained in the TS, to the sum of the weights of the entries $e'$ in the TS matching with $t$.

It is worth noting that the probability distribution depends on weights of all matching entries stored in the TS. Since all matching entries are contained in the partition of TS identified by the partition field of the template, also distributed implementations of the shared space can address the search of matching entries to specific locations by exploiting control fields. More in detail, to compute the probability distribution can be an expensive task; in order to reduce the computational cost of this phase, we can determine the multiset $TS' \subseteq TS$ containing all the entries $e$ such that $(e, w) \in TS$, for some $w \in Weight$, that may match the template, thus avoiding to consider entries in the TS that certainly do not match the template. To improve the performance of this phase the control fields used by the template can be exploited; for instance $TS'$ could be the partition of TS identified by the partition field used by the template. This set can be furtherly refined by exploiting, for example, a simple inequality test between entries and templates that says if certainly the entry do no match the template (e.g., it has not the same number of data fields or some fields have not the same value of the corresponding one in the template). This idea can be exploited, in a rather simple manner, also in distributed implementations of the shared space, indeed they could exploit partition fields to physically partition the TS, thus restricting the set of locations in which to perform the search.

Differently from control fields, we consider the weight as an attribute of the entry instead of as a possible additional data field. This is because weights are used to express the probability of the entries to be accessed and do not concern the content of the entries. Besides, we have not included in read and removal operations the possibility of indicating a specific weight of the matching entry,
because the goal is to properly distribute the accesses to the matching entries according to the associated weights. For example, when the shared space is used to implement the Web service registry mechanism the aim is to distribute the workload among the different implementations of the same task. Let us remark that this decision does not limit the expressive power because by using control fields it is possible to provide some users with better access to the discovery of Web services. More precisely, the access to some Web services can be reserved to a set of privileged users, augmenting (with respect to general users) their probability to find a Web service with an high quality of the service.

7.3.1 Implementation

A prototypal implementation based on that presented in Chapter 4, that implements a centralized space, has been developed. From the point of view of the interaction modalities and of the exchanged parameters described by the WSDL document the new description differs only in the entry structure that now contains also a new tag weight (that contains an integer value), that is reported in Section A.3. The tag weight represents the weight value associated with the entry; in order to reduce the changes in the WSDL document, we have opted for including the weight value within the entry structure. The weight, together with control fields, is not contained in the return value of data-retrieval operations.

In order to improve the performance of the matching search phase, as described previously, we exploit partitions to limit the space of possible matching entries. On the other hand, the complexity we add can be view as the price we must pay for introducing a resource distribution mechanism in the data-retrieval mechanism.
7.4 Supporting Web services workload distribution with probabilistic SecSpaces

Technologies for implementing Web service applications must be carefully developed so to provide critical guarantees to the consumers and to the suppliers of the services. In particular it is fundamental that such technologies ensure not only the security of transactions, but also an adequate quality of service, i.e. the availability of the service should be guaranteed to all users. We present a solution to this problem in the context of the emerging technology of Web services.

The Web services technology provides standard mechanisms for describing the interface and the services available on the Web, as well as protocols for locating such services and invoking them. In particular each Web service has an associated WSDL (Web Services Description Language) document which describes how it works and how to invoke it (via its “physical” address). Such a document is registered at an UDDI (Universal Description, Discovery and Integration) server that provides a discovery service for the WSDL descriptions.

Unfortunately, the current implementations of the Web services technology are based on a very primitive discovery protocol (the UDDI) that assumes that:

(i) Web services are located once and for all (at compile/design time) before being invoked from other Web services, and
(ii) there is no abstract/semantical concept of task performed by a certain Web service: several Web services may provide the same task by means of different implementations (e.g. by using different languages/algorithms) or replicas of the same implementation.

As a matter of fact this prevents an effective use of the current Web services technology for complex applications: since a run-time discovery of Web services is not allowed, it is not possible to take into account reconfigurations of the environment (i.e. Web services that appear and disappear after compile/design) and to manage distribution of the workload among several Web services implementing the same task. In other words UDDI supports discovery operations such as “provide me with a link to a specific kind of resource” while it is not expres-
sive enough for supporting queries such as “provide me with a link to a generic kind of resource performing a specific task in such a way that the overall resource usage is balanced”.

In Section 7.5 we present a solution that provides discovery of Web services at run-time and introduces the notion of abstract task in Web service registries. On this basis, we also provide the new discovery protocol with security access control mechanisms and a mechanism for distributing service invocations among several Web services implementing (at different efficiency levels) the same task.

The basic idea for providing run-time discovery of Web services is to use a discovery protocol which is itself a (statically located) Web service. In particular we resort to a particular kind of technology which, together with Web service choreography, is currently one of the hottest topic of research in the Web service community: Web services exploited at run-time by other Web services as coordination spaces. These services can be used in order to store and retrieve the information needed for managing the collaboration among Web services that are willing to cooperate (in our case registry information: WSDL descriptions of Web services associated with tasks). Some proposals have been already defined, see e.g. Ruple [Sof], WSSecSpaces, and the proposal by Álvarez et al [aBMM+03].

All these proposals are based on the notion of loosely coupled interaction obtained via generative communication [Gel85]: the coordination among the Web services does not occur via a direct communication, but it is mediated by the coordination space that is a shared repository of data. A sender communicates with one or more receivers writing data in the repository (e.g. the registration of a new Web service in our case); a receiver can read or consume these data from the space indicating with a pattern the kind of datum he is interested in (e.g. the discovery or the deregistration of a Web service implementing a given task in our case).

Moreover, some coordination spaces such as Ruple [Sof] and WSSecSpaces, also support security access control to the data inside the shared repository. In particular WSSecSpaces, which has a finer grained access permission mechanism w.r.t. [Sof], makes it possible to discriminate among the write, read and
consume access permission at the level of the single datum, hence to authenti-
cate/identify the producer of a datum or its reader/consumer. In the case of the
registry service these mechanisms allow us to model the fact that only allowed
agents, can register or deregister their entries from the registry. On the contrary
the retrieval (read) operation can be set to have a different (more free) access
modality. As far as the read operation is concerned, we can also model security
access control to registered Web services: they may be reserved to different classes
of users (e.g. staff of a factory providing services as opposed to the customers).

The proposal we are going to present exploits the SecSpaces model with
probabilistic retrieval of data in the repository. More precisely, we assume that the
coordination primitives are supplied by a statically located Web service that, in
our case, is the probabilistic version of WS_SecSpaces. We use this new feature to
model a discovery protocol which guarantees a rather concrete property related
to the quality of service: a balancing in the workload among different Web ser-
dvices executing the same task (possibly at different levels of efficiency). Note that,
in this respect, it is crucial for the discovery protocol to be based on a probabilistic
mechanism. Web services register themselves at the registry service by introduc-
ing data in the shared repository, while clients discover the availability of Web
services implementing tasks by performing data retrieval operations. Accord-
ing to this pattern of interaction, in the case several implementations for a given
task are available, one must be chosen. If the choice is just non-deterministic (as
in the other Linda-like coordination languages), there is no guarantee that the
invocations are uniformly distributed among the available servers. Given this
new probabilistic access mechanism, it is possible to obtain a registry service that
behaves in a convenient way. The efficiency level of a Web service is somehow
quantified and an appropriate weight is associated to the corresponding entry in
the registry service. When the clients perform the discovery of tasks executing
data retrieval operations, the underlying probabilistic access mechanism guaran-
tees a balanced distribution of the returned data (Web services implementing the
task).
7.5 A Web service registry guaranteeing Security and Quality of Service

In this section we propose an implementation of a Web service registry that provides clients with the following features: i) to discover Web services; ii) to register new Web services in the registry; iii) to remove a Web service from the registry. We take advantage of the extended SecSpaces presented in Section 7.3. More precisely, the three phases described above can be mapped into the three operations provided by SecSpaces: i) rd operations are used to discover services; ii) out operations to register services, and iii) in operations to deregister a service or to update its weight.

In Section 7.5.1 we present the Web services discovery protocol that exploits weights to properly balance the workload of Web service invocations. We also describe how control fields can be exploited to restrict the access to Web services. Section 7.5.2 describes how to register, deregister or update the information about Web services. Furthermore, we show how, by discriminating between the rd and the in access permissions, we make it possible to guarantee that only allowed agents (e.g., the Web services themselves) can register, remove or update entries from the registry. Finally, Section 7.5.3 describes an example of application that uses the proposed Web service registry (a software factory distributing its products to clients with different privileges).

7.5.1 Web service discovery

The UDDI [UDD] protocol supports registration and Web service discovery. More precisely, each agent can interact with a UDDI server that, given the name (and optionally a set of attributes) of a service returns the WSDL document associated to the requested Web service. The WSDL [CCMW] document contains all information about a Web service, such as the URL of the port that agents can use to access the service, the interaction modalities between requestor and Web service
(e.g., send-response) and the type of the argument passed to and received from the Web service (usually based on XML-Schema [Con]).

In the protocol that we are going to describe we assume that, similarly as in the standard UDDI protocol, each service is identified by a unique name. More precisely, we assume that the service indicates the kind of task performed; therefore, in general, many Web services supply the same task (e.g., the sort of a list of elements). Formally, let Task, ranged over by s, s′, . . . , be the set of tasks Web services can implement, WSDL, ranged over by ws, ws′, . . . , be the set of Web services. A registered Web service is represented by an entry in the TS; entries must have the tuples of data structured as follows (the range of values or the structure of control fields is left unchanged):

\[ d = \langle s, ws \rangle, s \in \text{Task and } ws \in \text{WSDL}, \]

where s indicates the task the Web service described by the document ws supplies.

The set of registered Web services is represented by the set of entries (with an associated weight) stored in the TS. To discover a Web service (i.e., to obtain its WSDL), an agent must perform a rd operation. For instance, an agent that is willing to discover a Web service supplying task s performs rd(\langle s, \text{null} \rangle[^#]); the return value will contain, in the second field, the WSDL associated to a Web service supplying task s. In this case the discovery is performed in the default space (default field identifiers # and ? are used), but, as it will be discussed in the following section, the discovery can be addressed also in reserved subspaces of the TS.

It is worth noting that in WSSecSpaces we exploit XML technologies to define entries (and templates) as well as to implement the matching evaluation between data fields. Therefore, our model does not preclude the discovery of a Web service satisfying a particular kind of data in the WSDL field of the tuple.

Finally, in our setting, the weights associated to each entry in the TS repre-
sent the quality level of the Web service associated to that entry, e.g., the performance degree of the Web service. In this chapter we do not discuss the mechanism/principle used to assign weights to Web services (e.g., based on the frequency of provided services): we consider the study of the methodology to establish the weights an orthogonal problem that can be tackled separately. We just assume that this values are determined by an external entity and that they can also change during the evolution of the system by invoking update operations (that we will define in the following) on the registry. The use of weights in the Web service discover enables a workload distribution of the Web service invocations that ensures some kind of quality of service requirement (depending on the meaning of the weights attributes).

Limiting the access to Web services

In several applications for open systems it is fundamental to allow the execution of specific tasks only to a subset of privileged users (see also Section 7.5.3), or to reserve a Web service to a specific set of users. Control fields, that provide a manner to implement access control mechanisms for read and removal operations, can be used to satisfy these goals.

Let us consider, as an example, a simple case in which there is a service (identified by task $s$) that manages the booking of airline flights that is used by Internet users and by booking agencies. In order to allow a more efficient service to agencies, one or more Web services supplying task $s$ are reserved for them; this can be done by publishing the entry representing those Web services in a partition identified by $c_s$, known only by the booking agencies. Observed that, from the numerical viewpoint, it is reasonable to presume the booking agencies to be fewer than users, the idea is to reserve a subset of the Web services to the agencies only: in this way we should ensure that the efficiency of the reserved service is better than the one experienced by the Internet users.
7.5.2 Registration and deregistration of Web services

The Web service discovery phase commented in the previous Section assumes a specific structure of the entries. In order to be coherent with this assumption, the registration phase of a Web service corresponds to an output of the entry containing the information about the service. More precisely, a Web service, say \( WS_A \), described by a WSDL document \( ws \in WSDL \) and providing task \( s \in Task \) that is willing to register, performs the following operation:

\[
\text{out}((s, ws)_{[c], [k], w}),
\]

where \( c, c' \in \text{Partition} \), \( k, k' \in A\text{Partition} \) and \( w \in \text{Weight} \).

Clearly, as previously discussed, the choice of control field values affects the visibility of the described Web service. Furthermore, the use of control fields indirectly provides also a trust measure of the registered Web services. Indeed, any agent can write into the public space of the TS and, therefore, any agent can register malicious or inexistent Web services in that space. Control fields can be exploited to register Web services in reserved spaces where it is assumed that agents having access are trusted. Other aspects related to the use of control fields will be discussed in the following.

The deregistration of a Web service can be done by simply removing from the TS the entry representing that Web service. For example, to deregister the service \( WS_A \) a \( \text{in} \) operation is performed, more precisely:

\[
\text{in}((s, ws)_{[c']}, [k], w').
\]

The update of a Web service in the registry consists of a removal operation followed by the output of an entry (with an associated weight) containing the updated information. For example, to update the weight of the Web service \( WS_A \) to a new value, say \( w' \), we need to perform
\[ \text{in}(\langle s, \text{ws}\rangle_{\mathcal{Cl}^I_{[k^I]}}) \text{ followed by out}(\langle s, \text{ws}\rangle_{\mathcal{Cl}^I_{[k^I]_{\text{lin}}}}, w'). \]

Finally, following the same idea used to update the weight, we can define a similar procedure to move a Web service from one subspace of TS to another one, e.g., to improve the performance of the Web services having access in the latter.

In order to guarantee the consistency of the published Web services, i.e. to allow only to the Web service maintainer (e.g., the Web service itself) the deregistration or the update of a registered Web service, we present a possible manner to limit the access to the entries that exploits the access permission for operations. As explained in detail in Chapter 2, if the Web service keeps secret the access keys (partition and asymmetric partition identifiers) for operations, only the Web service can remove (and then update) the service from the registry. Let us consider, as an example, a Web service registered by performing
\[ \text{out}(\langle s, \text{ws}\rangle_{\mathcal{Cl}^I_{[k^I]_{\text{lin}}}}, w). \] If \( k^I \) is known only by the Web service itself, it is the only one allowed to perform the update of the weight, because the knowledge of \( k^I \) is needed; indeed, the procedure to be performed for updating the weight to the value \( w' \in \text{Weight} \) is an \[ \text{in}(\langle s, \text{ws}\rangle_{[k^I]}) \text{ followed by out}(\langle s, \text{ws}\rangle_{\mathcal{Cl}^I_{[k^I]_{\text{lin}}}}, w'). \]

### 7.5.3 An Example of Application: A Software Factory

The effectiveness of the proposed solution is evaluated by exploiting it in a real application. Let us consider the case of an application that offers distinct sets of services for distinct categories of users. For instance, a software factory offering products to different users, e.g. partners (or privileged consumers) and generic consumers. Consumers can access the download service of the distributed products releases, while partners can also access the products releases not yet distributed (because, e.g., subject to testing phase). Moreover, in order to guarantee faster downloads, some Web services (distributed, e.g., over the Internet) are reserved to the partners. In this case, we have to support both separate allocation of Web services to different categories and a limited access to some Web services (i.e. those providing the download of unstable software releases).
Let $s$ (resp. $s'$) be the task offering the download of distributed releases (resp. unstable releases), $SU = \{ws_1, \ldots, ws_n \mid ws_i \in WSDL, 1 \leq i \leq n\}$ and $SP = \{pws_1, \ldots, pws_m \mid psw_i \in WSDL, 1 \leq i \leq m\}$ be the set of Web services providing task $s$ for generic consumers and partners, respectively, and $SP' = \{pws'_1, \ldots, pws'_l \mid pws'_i \in WSDL, 1 \leq i \leq l\}$ be the set of Web services supplying task $s'$. We also define weight be a function that, given a WSDL identifying a Web service, returns its weight. Let $c, c' \in \text{Partition}$ be the partition value identifying the space reserved for consumers and for partners, respectively. We also assume that $c$ is known only by the consumers (that can acquire its value, e.g., after a registration phase), while the knowledge of $c'$ is limited to the partners.

The registration procedure of Web services composing the application follows:

- $\text{out}(\langle s, ws \rangle^{[c]_{\text{rd}}}_{\text{declin}})$, weight(ws), for any $ws \in SU$ (registration of Web services for consumers).
- $\text{out}(\langle s, ws \rangle^{[c']_{\text{rd}}}_{\text{declin}})$, weight(ws), for any $ws \in SP$ (registration of Web services supplying task $s$ for partners).
- $\text{out}(\langle s', ws \rangle^{[c']_{\text{rd}}}_{\text{declin}})$, weight(ws), for any $ws \in SP'$ (registration of Web services supplying task $s'$ for partners).

To discover these three classes of Web services a rd operation is to be performed, more precisely:

- $\text{rd}(\langle s, \text{null} \rangle^{[c]}_{\text{rd}})$, to discover a Web service supplying task $s$ for consumers;
- $\text{rd}(\langle s, \text{null} \rangle^{[c']}_{\text{rd}})$, to discover a Web service supplying task $s$ for partners;
- $\text{rd}(\langle s', \text{null} \rangle^{[c']}_{\text{rd}})$, to discover a Web service supplying task $s'$ for partners.

It is worth noting that the balanced access to the download services is guaranteed by the weights weight(ws) associated to each Web service $ws$. Only partners can perform the latter two operations because the knowledge of $c'$ is needed, satisfying so the goals described above.
Finally, asymmetric control fields (above left to default values) can be exploited to refine access permissions. For example, it can be reasonable to guarantee users that registered services are indeed provided by the software factory. This can be done by exploiting asymmetric partition fields associated to the rd access permission. The fact that the entity that wrote the entry in the tuple space owned the key stored in the rd asymmetric partition field authenticates its identity: users (that own the “public” key) know that the only entity possessing the “private” co-key is the software factory.

7.6 Conclusion

In this chapter we have described how to combine priorities and probabilities introduced in the previous chapter with the SecSpaces model. In particular, we have presented a model supporting dynamic name-based priorities and another one supporting probabilistic access to the entries. The two aspects are orthogonal, thus to merge the two proposals is straightforward.

As an example of application, the probabilistic SecSpaces has been exploited to present a possible Web services registry providing a discovery protocol supporting: i) the discovery of Web services at run-time that takes into account the modifications of the system; ii) quality of service (by exploiting weights on entries); iii) a mechanism to limit the access to Web services (by exploiting control fields). It is worth to remark that the proposed solution does not compete with the standard UDDI protocol but, rather, aims at covering the most critical requirements by integrating it with a coordination infrastructure.

To define an extending setting where we associate with entries more than one weight attribute can be easily done. In this way, for example, registered services can describe several indexes of quality of the service and the discovery phase can be parameterized on more than one degree of quality, allowing clients to address the discovery of Web services with specific peculiarities.
Appendix A

WSSecSpaces specifications

A.1 Entry and templates XML-schema

```xml
<?xml version='1.0' encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="entry">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="rd" type="ControlField" />
        <xsd:element name="in" type="ControlField" />
        <xsd:element name="dataField" type="DataField" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="template">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="apartition" type="APartition" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```
</xsd:element>
<xsd:element name="partition" type="xsd:string">
</xsd:element>
<xsd:element name="tdataField" type="TDataField">
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="ret">
<xsd:complexType>
<xsd:choice>
<xsd:element name="datafield" type="DataField">
</xsd:element>
<xsd:element name="noValue" type="xsd:string">
</xsd:element>
</xsd:choice>
</xsd:complexType>
</xsd:element>
<xsd:complexType name="ControlField">
<xsd:sequence>
<xsd:element name="apartition" type="APartition">
</xsd:element>
<xsd:element name="partition" type="xsd:string">
</xsd:element>
</xsd:sequence>
</xsd:complexType>
<xsd:complexType name="APartition">
<xsd:sequence>
<xsd:element name="plaintext" type="xsd:string">
</xsd:element>
</xsd:sequence>
</xsd:complexType>
}
A.2  WSDL of WSSecSpaces

<?xml version='1.0' encoding="UTF-8"?>
<definitions>
  targetNameSpace="urn:3950"
  xmlns="http://schemas.xmlsoap.org/wSDL/"
  xmlns:xsd="http://www.w3c.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wSDL/soap/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:tns="urn:3950"
  <import namespace="secS"
     location="http://cs.unibo.it/lucchi/SecSpaceCorePack/secS.xsd"/>
</import>

<types>
  <xsd:simpleType name="OutOp">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="out"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="InOp">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="in"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="RdOp">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="rd"/>
    </xsd:restriction>
  </xsd:simpleType>
</types>
<message name="sendEntry">
  <part type="secS:entry" name="sendEntry"/>
</message>

<message name="sendTemplate">
  <part type="secS:template" name="sendTemplate"/>
</message>

<message name="returnValue">
  <part type="secS:ret" name="returnValue"/>
</message>

<message name="Action">
  <part type="OutOp" name="outA"/>
  <part type="InOp" name="inA"/>
  <part type="RdOp" name="rdA"/>
</message>

<PortType name="theTupleSpace">
  <operation name="out">
    <input message="sendEntry"/>
  </input>
</operation>

<operation name="in">
  <input message="sendTemplate"/>
</input>
<output message="ReturnValue"/>
</output>
</operation>
<operation name="rd">
<input message="sendTemplate"/>
</input>
<output message="ReturnValue"/>
</output>
</operation>
</PortType>

<binding name="pgBindingWSSS" type="tns:theTupleSpace">
<soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
</soap:binding>
<soap:operation soapAction=""/>
</soap:operation>
<operation name="out">
<input>
<soap:body use="encoded"
    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
</soap:body>
<soap:header message="Action" part="outA" use="encoded"
    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
</soap:header>
</input>
</operation>
<operation name="in">
<soap:operation soapAction=""/>
</soap:operation>
<input>
  <soap:body use="encoded"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
</input>

<output>
  <soap:body use="encoded"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
</output>

<operation name="rd">
  <soap:operation soapAction="" />
</operation>

<input>
  <soap:body use="encoded"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
</input>

<output>
  <soap:body use="encoded"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
</output>

<soap:header message="Action" part="inA" use="encoded"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
</soap:header>
</operation>
</binding>

<service name="WSSecSpaces">
  <documentation>Web service implementing SecSpace primitives</documentation>
  <port name="secSpace" binding="pg:pgBindingWSSS">
    <soap:address location="https://sarastro.cs.unibo.it:8080/WSSecSpace/secSpace"/>
  </port>
</service>

A.3 Entry with weight

<xsd:element name="entry">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="weight" type="xsd:integer"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element>


[GLZ] Roberto Gorrieri, Roberto Lucchi, and Gianluigi Zavattaro. Supporting Secure Coordination in SecSpaces. Fundamenta Informaticae. Submitted for publication.


