XML Dataspaces for Mobile Agent Coordination

Giacomo Cabri, Letizia Leonardi, Franco Zambonelli
Dipartimento di Scienze dell'Ingegneria – Università di Modena e Reggio Emilia
Via Campi 213/b – 41100 Modena – ITALY
Phone: +39-059-376735 – Fax: +39-059-376799
E-mail: {giacomo.cabri, letizia.leonardi, franco.zambonelli}@unimo.it

Abstract

This paper presents XMARS, a programmable coordination architecture for Internet applications based on mobile agents. In XMARS, agents coordinate – both with each other and with their current execution environment – through programmable XML dataspaces, accessed by agents in a Linda-like fashion. This suits very well the characteristics of the Internet environment: on the one hand, it offers all the advantages of XML in terms of interoperability and standard representation of information; on the other hand, it enforces open and uncoupled interactions, as required by the dynamicity of the environment and by the mobility of the application components. In addition, coordination in XMARS is made more flexible and secure by the capability of programming the behaviour of the coordination media in reaction to the agents' accesses. An application example related to the management of on-line academic courses shows the suitability and the effectiveness of the XMARS architecture.

Keyword*: Internet Architectures, XML, Mobile Agents, Tuple Spaces, Interoperability*

1. Introduction

The growth of the Internet infrastructure and the pervasion of the WWW technology have changed the way the Internet is conceived and exploited. Far from considering the Internet either as a raw communication media or as a global information repository, the current challenge is to exploit the Internet as a globally distributed computing system, where any kind of computation on distributed data and coordination among distributed entities can be performed. However, deploying distributed applications in the Internet raises several problems, due to the intrinsic characteristics of the Internet itself. First, decentralised management and openness make the Internet populated by a multitude of heterogeneous entities (services and information sources) with which application components may be in need to interact, thus raising *interoperability* problems. Second, the intrinsic dynamicity of the scenario and the unreliability of Internet connections require suitable *programming paradigms* and *coordination technologies*, to facilitate the design and the execution of Internet applications.

With regard to *interoperability*, the incredible success of HTML has recently led the WWW consortium to the development of XML [W3Ca], a language for data representation which is likely to become a standard for interoperability in the Internet, due to the advantages in can provide in this context [KhaR97]. First, XML represents data in a familiar (HTML-like) tagged textual form, and explicitly separates the treatment of data from its representation. Unlike an HTML document, which expresses how a document must be processed and visualised by a browser, an XML document only specifies what the data structures are, and leaves any processing decision at the application level. For example, a browser must be instructed about how to visualise an XML document via an XSL style sheet [W3Cb]. This achieves the platform-independence required for the Internet and the well appreciated feature of human-readability. In addition, since the XML tag-set is freely extensible,

XML can be made capable of representing whatever kind of data and entity one is likely to find in the Internet: complex documents [CiaVM99], service interfaces and objects [GluTM99], communication protocols [NakY98], as well as agents [LanHO99]. These characteristics let us think that interoperability in the Internet will be information-oriented and based on XML, rather than service-oriented and based on CORBA [OMG97]. In fact, by focusing mainly on communication interoperability, CORBA falls short when complex data and documents are involved.

With regard to *programming paradigms* and *coordination technologies* for Internet applications, the most suitable solutions seem to be represented by mobile agents and Linda-like tuple spaces, respectively. On the one hand, a programming paradigm based on *mobile agents*, i.e., active and autonomous software entities that can dynamically plan their execution activities, thereby included the capability of transferring their execution across different execution environments (i.e., Internet sites), suits well Internet applications. In fact, mobile agents can help application designers in dealing with the intrinsic uncertainty that they can have about the target environment, as well as with the intrinsic dynamicity and unreliability of Internet sites and their connections [FugPV98, KarT98]. On the other hand, whenever mobility and dynamicity are involved, coordination models which forces a strict coupling between the interacting entities (such as peer-to-peer and client-server ones) forces odd design choices in applications and lead to inefficiency in execution. Therefore, we argue that fully uncoupled coordination models based on the tuple space concept [AhuCG86, PapA98], adopted for both inter-agent coordination and for making agents interact with their current execution environment, suits Internet applications based on mobile agents and leads to simpler application design [Auth99].

Putting all together, we have designed and implemented XMARS, a coordination architecture for mobile agents, which exploits the advantages of the XML language and of Linda-like coordination in the context of a more general Internet architecture based on XML. In XMARS, derived from the MARS coordination architecture [Auth98], agents coordinate – both with each other and with their current execution environment – through programmable XML dataspaces associated to each execution environment and accessed by agents in a Linda-like fashion, as if they were tuple spaces. This can provide several advantages in mobile agent applications. On the one hand, by exploiting XML as the base language for data representation, XMARS provides for a high-degree of interoperability among the multitude of heterogeneous information sources with which an agent may be in need of interact. On the other hand, by exploiting Linda-like coordination, XMARS enables a high-degree of uncoupling in interactions and suits the openness and the dynamicity of the Internet scenario. In addition, since the behaviour of the XML dataspaces in response of the accesses made to them by agents can be fully programmed [DenNO98], XMARS enables both environment-specific and application-specific coordination laws to be embedded in a dataspace, thus providing for more secure and flexible coordination activities. An application example in the area of agent-mediated management of on-line academic courses is assumed as a case study to show the effectiveness of the XMARS approach.

The paper is organised as follows. Section 2 discusses the architectural issues arising in the adoption of XML as a base for interoperability and proposes a general Internet architecture based on XML. Section 3 presents the XMARS architecture. Section 4 presents the application example. Section 5 discusses related work. Section 6 concludes the paper and outlines our future research work.

2. XML Architectures

XML is a promising technology to achieve interoperability in the Internet world. However, one should ask what architecture could be conceived so as to allow heterogeneous components to interoperate via XML dataspaces. In this section, starting from two simple and intuitive XML-based sample architectures, we will try to sketch a more general architecture, based on XML, which can

accommodate any kind of Internet services and information, as well as any kind of application level component, included mobile agents.

2.1 XML on the Web

Currently, the most natural and intuitive way to exploit XML is in the Web server, as a data representation format more flexible and powerful than HTML. In XML, like in HTML, data is recorded in a standard, textual, format that can be read and manipulated by several kinds of application. However, in HTML, tags explicitly commit applications to a specific use (i.e., a specific visualisation format) of the enclosed information. Instead, in XML, it is up to the application level to decide what to do (i.e., how to elaborate and visualise) enclosed information. Therefore, in the context of Web servers, XML enables a more modular approach to servers' design. At the lowest level, the server administrator can store its information in XML, adopting the most natural format for its data, and disregarding any issue related to data visualisation. At a higher level, the XML-coded information can be translated into HTML pages to be visualised by a standard HTML browser, on the basis of specific visualisation information specified in XSL [W3Cb]. In this case, a software layer is needed to translate data from XML to HTML, accordingly to XSL specifications, and to provide browsers with the normal interface of a HTTP server. In addition, the information level can also store XML information related to server management, to be read and interpreted by the server during its activities, as it is for example defined by the Channel Definition Format [CDF97], and exploited by Microsoft Active Channels.

By trying to generalise the above example, one could think at the XML information in the server as originated by a different information source and made available in XML format for use by the XML server (as it can considered an HTTP server capable of translating XML to HTML on the basis of XSL). Let us consider a DBMS that stores information using a proprietary representation. The most efficient way to make this information available on the Web is to provide a software layer that dynamically queries the database and produces its results in a Web accessible format. This is what happens in several Web servers, where a CGI application is provided to access a DBMS and translate the results in HTML. Also in this case, a more modular approach would first translate the information produced by the database in XML format, to be further translated by the XML server in HTML, when to be retrieved by a browser.

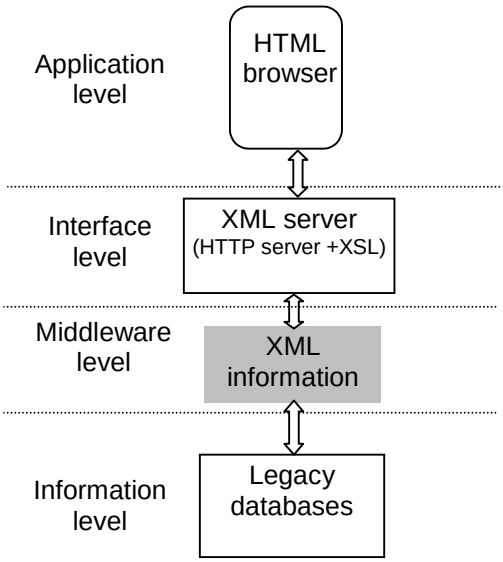

Figure 1. An XML architecture for the Web

2.2 Towards a General XML Architecture

The previous example shows how exploiting XML in the Web naturally leads to a multi-layered architecture (see figure 1). At the *application level*, the browser must be allowed to

elaborate information in HTML format. Therefore, if a browser is not an XML-enabled one, an XML server (that is, an HTTP server capable of transforming XML documents into HTML ones, accordingly to XSL specifications) must be provided at the *interface level* for interfacing the browser with the XML dataspace. The XML dataspace acts as a *middleware level* for the data stored in a DBMS at the lowest *information level*. Whenever the information on a site is not already stored in XML format (in which case the middleware and the information level collapse in a single level), data is translated from/to the information level up to the middleware level.

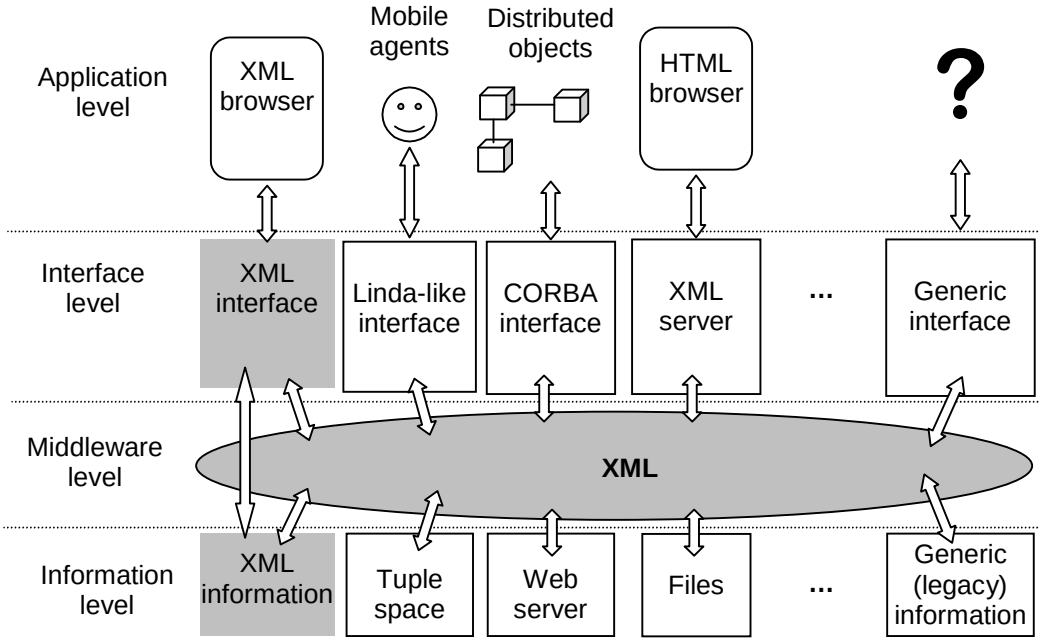

Figure 2. A general architecture based on XML

The identified four-level architecture may be considered as a more general architecture for Internet applications (see Figure 2). In the information level, we can take into account the presence of any kind of data stored in whatever kind of format: bare files, tuples in a tuple space, objects, service interfaces. Provided that all this data can be represented in XML, the middleware level can furnish the necessary tool to transform this data in XML format and viceversa, and store (or dynamically produce) them into the XML dataspace. It is expected that the emergence of XML as a standard will make these kinds of tools widely available. The interface level is in charge of publishing the information in the formats requested by the different application components and/or according to specific protocol/coordination models. Different components may be present at this level, depending on the variety of different application components that may be in need of accessing the XML dataspace from the application level. Such components may include XML browsers, CORBA interfaces for enabling distributed object applications to interact with the XML dataspaces, or Linda-like interfaces for enabling mobile agent coordination. It is expected that the increasing diffusion of XML will also increase the capability of application components to directly read and elaborate on XML dataspaces, thus making it unnecessary any translation of data from XML to a specific application level format. Nevertheless, the interface level will maintain its role of implementing specific policies of access to the data. On the one hand, it has to implement the necessary policies to preserve the consistency of the XML data, from simple readers/writers policies up to complex transaction-based ones, if necessary. On the other hand, the interface level must implement the necessary access control policy to protect data from malicious access.

The above described architecture is very general and presents several advantages. First, it grants a high degree of interoperability, since different applications can access information accordingly to their own interaction model. Second, it permits to access in the XML format whatever kind of information, translating the information on-demand and giving the possibility of

maintaining the original format. Third, the architecture exhibits a high degree of scalability, since different kinds of information format or interface can be added by designing an appropriate XML translator.

In the following, we will restrict our focus to mobile agents applications, and on the provision of a Linda-like interface to mobile agents.

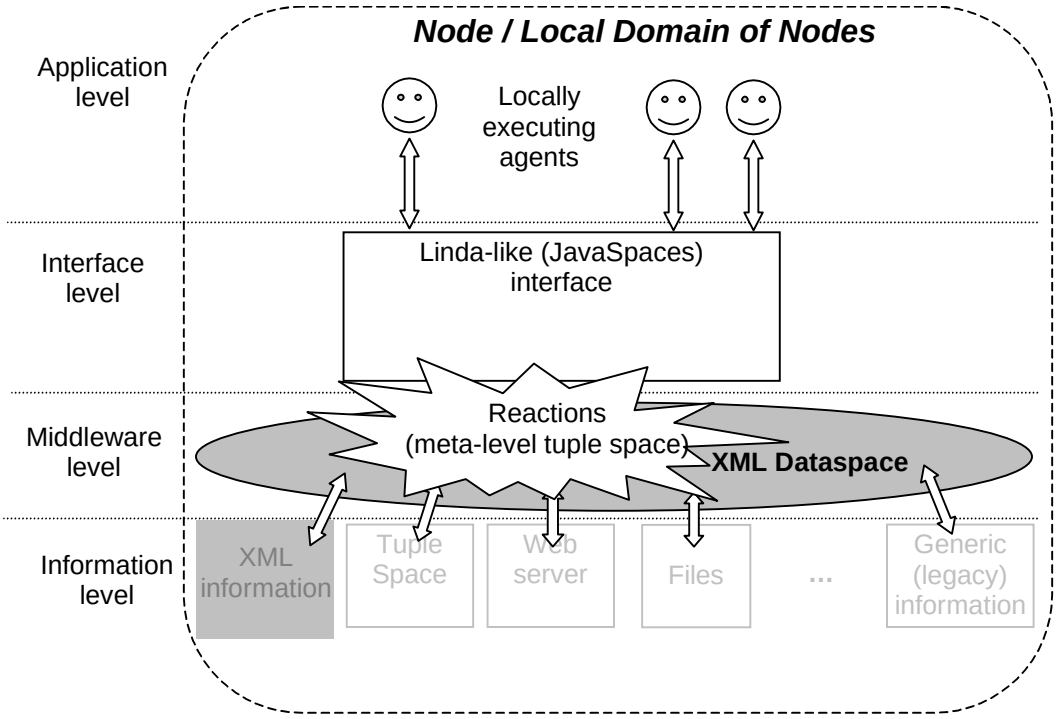

Figure 3. The XMARS architecture

3. The XMARS Coordination Architecture

In mobile agent applications, the adoption of a Linda-like coordination model is the most suitable solution for both inter-agent and agent-to-execution environment coordination [Auth99, OmiZ99]. In the context of the general XML architecture described in the previous section, accommodating a Linda-like coordination style for mobile agents amounts at: *(i)* integrating specific architectural solutions tuned to mobile agent applications; *(ii)* providing a Linda-like interface for enabling agent access to XML dataspaces; *(iii)* enabling the association of specific computational activities to the accesses performed by agents, thus defining a programmable tuple space model.

In this section, we describe the XMARS coordination architecture for mobile agents (see Figure 3) and show how the above issues have found appropriate solutions, via: *(i)* an architecture which enforces locality in the accesses; *(ii)* the implementation of the standard JavaSpaces interface and *(iii)* a simple yet effective programmable tuple space model based on meta-level tuple spaces.

As a general note, we emphasise that XMARS does not implement a whole new Java agent system. Instead, it has been designed to complement the functionality of already available agent systems, and it is not bound to any specific implementation: it can be associated to different Java-based mobile agent systems with only a slight extension. In addition, we feel it should not be difficult to adapt the current implementation in order to allow different application-level entities, in addition to mobile agents, to exploit XMARS for the coordination of their activities.

3.1 The XMARS Architecture

Since mobile agents travel the network in the effort of enforcing locality in the access to the needed resources, the architecture needs to somehow facilitate the agents' efforts. To this end, a coordination architecture for mobile agents (whether based on XML dataspaces or not) must provide a multiplicity of independent tuple spaces, to be accessed by agents in a local way. This means that XML dataspaces have to be considered local resources associated to a node, or at most to a local domain of nodes.

XMARS enforces the concept of locality in interactions: an Internet node must define its own XML dataspace and the associated Linda-like interface. When an agent arrives on a node, it is automatically provided with a reference to the local XMARS tuple space interface associated to that node. Then, it can use this reference to access the XML dataspace in a Linda-like fashion, i.e, by reading, extracting, inserting fragments of XML data into the dataspace, as if they were tuples. In addition, a local domain of nodes (e.g., a local network) can federate and implement a single XML dataspace. All the agents executing in nodes of the domain are automatically provided with a XMARS interface reference, with which they can access the XML dataspace transparently from anywhere in the domain.

3.2 The XMARS Interface

XMARS adopts a JavaSpaces [JS98] compliant interface (see Figure 4). Therefore, as in JavaSpaces, an XMARS tuple space assumes the form of a Java object making available the following three operations for accessing the tuple space:

- read, which retrieves a tuple matching a given template;
- take, which extracts the matching tuple from the tuple space;
- write, which puts a tuple in the tuple space.

Two additional operations, readAll and takeAll, not present in the JavaSpaces interface, have been added to the XMARS interface to avoid agents being forced to perform several reads/takes to retrieve all the needed information, with the risk of retrieving the same tuples several times. This is a well-known problem of tuple space model, due to the non-determinism in the selection of a tuple among multiple matching ones.

```
public interface XMARS extends JavaSpace
{
// interface methods inherited from JavaSpace
// Lease write(Entry e, Transaction txn, long lease);        // put a tuple into the space
// Entry read(Entry tmpl, Transaction txn, long timeout); // read a matching tuple from the space
// Entry take(Entry tmpl, Transaction txn, long timeout); // extract a matching tuple from the space

// methods added by MARS and not present in the JavaSpace interface
Vector  readAll(Entry tmpl, Transaction txn, long timeout);          // read all matching tuples
Vector takeAll(Entry tmpl, Transaction txn, long timeout);           // extract all matching tuples
}
```

Figure 4. The XMARS interface

For all the operations, the txn parameter can specify a transaction the operation belongs to. The timeout parameter of read, take, readAll and takeAll, specifies the time to wait before the operation returns a null value if no matching tuple is found: while NO_WAIT means to return immediately, 0 means to wait indefinitely. The lease parameter of the write operation sets the lifetime of the written tuple.

Our choice of implementing JavaSpaces interface rather than defining a new "XML-oriented" Linda-like interface, has been mainly driven by the fact that the JavaSpaces technology is likely to have a great impact in the context of distributed Java applications. Furthermore, we think that the JavaSpaces choice better suits the object-orientation of Java agents, by making them manage object tuples rather than XML fragments. In fact, from the agents' point of view, XMARS

tuples are Java objects whose instance variables represent the tuple fields. The interface operations are in charge of translating the object representation of tuples to the corresponding XML representation (write operation) and viceversa (read, take, readAll, takeAll operations), as well as of handling the insertion/removal of tuples from the documents of the XML dataspace.

In JavaSpaces, and therefore in XMARS, tuple management requires that tuple classes implement the Entry interface. The easiest way to do this is to derive a tuple class from the AbstractEntry class (that defines the basic tuple properties by implementing the Entry interface) and to define, as instance variables, the specific tuple fields. Each field of the tuple is a reference to an object that can also represent primitive data type (*wrapper* objects in the Java terminology). In addition, in XMARS, a tuple class must have a static private field that specifies the DTD that describes the structure of the XML documents corresponding to the instances of such class. In fact, in XMARS the DTDs correspond to the tuple classes, as well as the XML documents correspond to the tuple objects. An example of tuple class is shown in figure 5, while figure 6 shows how an agent can use this class to access the XML dataspace.

```
class _infoN extends AbstractEntry {      // AbstractEntry: generic tuple
        //class field
        static private final URL DTDfile = new URL("http://mysite/myDTD.dtd");
        // tuple fields
        public Integer f1;
        public String f2;
        public String f3;
        public Integer f4;
}
```

Figure 5. Example of tuple class

```
...
_infoN t = new _infoN();
t.f2 = "foo";
t.f3 = "*bl*";
t.f4 = 17;
myEntry result = space.read(t, new Transaction(null), NO_WAIT);
...
```

Figure 6. Fragment of code of an agent

A tuple corresponds to an element of an XML document. In particular:

- the name of the class, which must begin with the underscore character corresponds, once deleted the initial underscore, to the name of the tag defining the XML element;
- the names of instance variables correspond, orderly, to the name of the tags enclosed in the element;
- the values of the instance variables correspond, orderly, to the data enclosed in the corresponding tags.

Names of classes and instance variables can be dynamically acquired via Java reflection. For example, the _infoN tuple class defined in figure 5, corresponds to an <infoN>…</infoN> element in the XML document of figure 7. The translation of a tuple to/from the corresponding representation as an XML element is automatically performed by XMARS upon invocation of one of the interface operations. This also includes the translation in a textual form of non-string fields, such as Integer and Float ones.

The pattern matching currently implemented in XMARS uses a textual comparison between the XML elements. A template tuple in an input operation can contain *formal values* (i.e., not defined), which can match with any value in the corresponding XML element, and *actual values* (i.e., with a well-defined value). In the case of string fields, *partially defined values* can be expressed by exploiting wild cards ("*" and "?" only, in the current implementation). When an input

operation is invoked by an agent, XMARS performs a search in the XML dataspace, to find one element in a XML document such that:

1. the DTD used by the document is the one specified in the static private field of the template tuple;
2. the tuple translated in XML format corresponds to at least one element in the document
3. the values of the defined (or partially defined) fields in the tuple correspond to the values in the tags of the element.

Figure 7 shows (in gray background) the fragment of an XML document that can be returned, in the form of a tuple, by the read operation requested in figure 6. The agent initialises the fields f2, f3 and f4, maintaining f1 as formal. Match occurs because the values of the fields f2 and f4 are the same of those specified in the template tuple by the agent, and because the value of the field f3 matches the regular expression in the field f3 of the tuple. The same XML fragment would have been extracted from the document if the agent would have performed a take operation with the same template tuple.

```
<?XML version="1.0"?>
<!DOCTYPE myEntry SYSTEM “http://mysite/myDTD.dtd”>

<infoN>
        <f1>3</f1>
        <f2>foo</f2>
        <f3>blahblah</f3>
        <f4>17</f4>
</infoN>
<infoN>
        <f1>5</f1>
        <f2>foo2</f2>
        <f3>bar</f3>
        <f4>23</f4>
</infoN>
```

Figure 7. Tuples as XML elements

The current implementation of the XMARS interface, although operative, still suffers of some limitations, which are being addressed at the time of writing. First, the interface still lacks the capacity of managing in an appropriate way all of the components and structures that can be found in XML documents. In particular: *(i)* the current implementation considers the attributes of a tag as additional tags, thus lacking in realising a perfect one-to-one correspondence between XML elements and Entry objects; *(ii)* the implementation does not handle *XML namespaces*, necessary to avoid conflicts in the tags' names and, therefore, to correctly implement the pattern-matching mechanism. Second, synchronisation of concurrent accesses is based on a MR/SW (Multiple Readers/Single Writers) policy applied at the level of single XML documents. This choice promotes simplicity and also allows any other application-level entity to access an XML dataspace in concurrency with mobile agents, provided that it conforms to the same policy. However, the MR/SW policy at the level of single documents may not be the most appropriate choice when multiple agents are in need of performing complex transactions over different documents, or when multiple agents are in need of performing concurrent operations on different parts of the same XML documents. Therefore, we are currently evaluating whether different synchronisation policies can be conceived and can coexist with the MR/SW in a dataspace. A third direction of improvement relates to security control. Currently, the mechanisms for controlling the accesses to the XML dataspace are based on Access Control Lists associated to each XML document, to specify which agents can read, write, or extract elements (i.e., tuples) from the document. This kind of access control may have coarse granularity when mobile agents are in need of working at the level of single tuples.

3.3 The XMARS Reactive Model

The behaviour of XMARS can be programmed, both by administrators and by mobile agents, via the installation of *reactions* associated to specific access events, which are triggered when the corresponding events occur. Such reactions can modify the result of the operations they are associated with, can manipulate the content of the XML dataspace, and can access whatever kind of external entity they are in need of accessing.

The introduction of a programmable tuple space model (whether based on XML dataspaces or not) provides for much greater flexibility and control in interactions than the raw Linda model [Auth99, OmiZ99]. A site administrator can program reactions to monitor the access events to the local resources and, in need, to issue specific actions to preserve its resources from malicious accesses. Reactions can be used to implement a dynamic dataspace model, in which the data is not statically stored in the dataspace but, instead, is dynamically produced on-demand and is possibly originated from different sources. In the context of XML dataspaces, and with reference to the general XML architecture sketched in section 2, reactions can thus act as a bridge between the information level and the XML middleware level (see Figure 3). Dynamic production of tuples also enables a simple data-oriented way of accessing services on a site: the attempt to read a specific tuple by an agent can trigger the execution of a local service, in charge of producing the required tuple as a result. In this direction, reactions can be used to establish and control session-based communications between agents. Consequently, in the context of XML dataspaces, which are likely to be accessed by entities other than mobile agents, we also envision the possibility of exploiting reactions to control the interactions between entities that are heterogeneous in terms of supported coordination models.

Further advantages could be provided by giving application agents the capability of defining their own coordination rules: agents can carry along the code of the reactions implementing application-specific coordination policies, and install them in the tuple spaces of the sites visited. This achieves a sharp separation of concerns between algorithmic and coordination issues [GelC92]: the agents are in charge of embodying the algorithms to solve the problems; the reactions represent the application-specific coordination rules. This can both reduce the agent complexity and simplify the global application design, as shown in [Auth99].

In the implementation of a programmable tuple space model, XMARS currently exploit the implementation of the MARS reactive model [Auth98]. Reactions are stateful objects with a method (named "reaction") whose code represents the core of the reaction, and are associated to specific access events on the basis of the three components that characterise the access event: *tuple item (T), operation type (O)* and *agent identity (I)*. Therefore, the association of a reaction to an access event is then represented via a 4-*ple (Rct, T, O, I)*: the reaction method of the object *Rct* (i.e., the reaction itself) is executed when an agent with identity *I* invokes the operation *O* on a tuple matching *T*. In this perspective, the association of reactions to tuples can be considered as dealt with *meta-level tuples* enclosed in a *meta-level tuple space*, which has to follow associative mechanisms similar to the one of the base-level tuple space. Putting and extracting tuples from the meta-level tuple space provide installing and de-installing reactions. A meta-level 4-*ple* (possibly with some non-defined values) associates the reaction to all the accesses that matches that 4-*ple*. A match in the meta-level triggers the corresponding reaction. For example, a meta-level 4-*ple* (ReactionObj, null, read, null) associates the reaction of ReactionObj to all read operations, disregarding both the peculiar tuple content and the agent identity. Analogously, one can associate reactions to a specific tuple, to all tuples, or to the tuples of a given class. The pattern-matching mechanism in the meta-level tuple space is activated for any access to the "base-level" tuple space to check for the presence of reactions to be executed.

The reaction method has to be defined according to the following prototype:

```
public Entry reaction(XMARS s, Entry Fe, Operation Op, Identity Id)
```

The parameters represent, orderly: the reference to the local XMARS space, the reference to the tuple resulting from the matching mechanism issued by the associated operation, the operation

itself, and the identity of the invoking agent. The code of the reaction has access to the base-level tuple space and to the meta-level one and can perform any kind of operation on them (tuple space access operations performed within the code of a reaction do not issue any reaction, to avoid endless recursions). As a consequence, the behaviour of the reaction can depend both on the actual content of the tuple space and on past access events. Also, having the availability of the result of the matching mechanism and of the associated operation, the reaction can also influence the semantics of operations and, for example, can return to specific invoking agents different tuples than the ones resulting from the raw, stateless and less flexible, Linda-like matching mechanisms.

The current security mechanisms defined for the meta-level tuple space are, again, based on Access Control Lists, to specify which agents have the rights to install which kinds of reactions in an XML dataspace. For instance, a manager agent may be allowed to install whatever kind of reaction; a generic application agent can only install reactions to be triggered by other agents of the same applications. Also in this case, we feel that the above security model is too coarse-grained to satisfy the needs of both administrators and application agents, and its refinement is an in-progress work direction. As another limitation, the current implementation of the XMARS reactive model, which fully re-use the MARS implementation [Auth98], does not realise the meta-level tuple space as an XML dataspace. Instead, it stores meta-level tuples in an object form and, consequently, requires the meta-level pattern-matching mechanisms to occur with application level tuples in their object form, too. It is our intention to upgrade the implementation of the meta-level tuple space so as to make it become an XML dataspace in its turn.

4. Application Example

In this section, we apply XMARS in the context of the management of on-line university courses. First, we introduce some simple examples related to information retrieval and management, with the goal of both clarifying the presented concepts and showing a few samples of XML documents and Java code. Then, we sketch a more complex example related to inter-agent coordination to illustrate the power of the XMARS programmable coordination model.

4.1 Information Retrieval and Management

Let us suppose different universities and academic departments federate to make the material of the courses available on-line, to allow both students to easily retrieve material of interest and teachers to easily upgrade it. To this end, they can agree on adopting XML as a standard data representation format for course material, and adopt a well-defined DTD for the corresponding XML documents. Figure 8 shows a sample of XML document containing information related to a course. This include general information about the course, as well as specific information for each of the lessons of the course, such as abstract and suggested readings.

With reference to the general architecture described in section 2, the XML representation of course material defines an XML dataspace that could either derive from a translation of previously available material stored at the information level in a different format, or be written from scratch in XML if no material were previously available. In addition, this material could be made available to application clients in different ways, i.e., by adopting different interfaces.

As a first, simple, example related to information retrieval, let us consider a student that wants to acquire information related to a course or to a specific lesson. If the academic federation makes available XML servers on their sites, (s)he can simply adopt a browser to navigate trough the XML dataspaces of the federation and analyse their content. However, this search activities can be rather time expensive and boring if the federation is a large one and the course material detailed and verbose. Therefore, if the sites of the federation can host mobile agents and make the XMARS interface available, students can decide to delegate the search activities to mobile agents. To this end, the agent must define (or have the availability of) the tuple classes representing the XML elements of interest. Figure 9 reports the definition of the tuple classes representing the XML

elements <course>, <lesson> and <reading>. Then, the agent can exploit the operations of the XMARS interface to retrieve the needed information in a simple and effective way. Figure 10 reports a simple code fragment of an agent that travels across the sites of the federation to retrieve information about those lessons containing the keyword "network" in the abstract.

```
<?XML version="1.0"?>
<!DOCTYPE CourseEntry SYSTEM “http://university.site/UnivCourse.dtd ”>

<course>
        <coursename>Computer Networks</coursename>
        <year>4</year>
        <semester>1</semester>
        <teacher>Jane Smith</teacher>

        <lesson>
                <lessonname>Introduction</lessonname>
                <lessonnumber>1</lessonnumber>
                <abstract>blah blah</abstract>
                <reading>
                        <authors>..</authors>
                        <title>..</title>
                        <book>..</book>
                        ….
                </reading>
                <reading>….</reading>
        </lesson>

        <lesson>
                <lessonname>Basic Protocols</lessonname>
                <lessonnumber>2</lessonnumber>
                <abstract>blah blah</abstract>
                <reading>….</reading>
        </lesson>
….
</course>
```

Figure 8. Example of an XML document describing a course

```
class _course extends AbstractEntry {
        static private URL DTDfile = new URL(“http://university.site/UnivCourse.dtd”);
        public String coursename;               // the name of the course
        public Integer year;                    // when it is scheduled: year
        public Integer semester;                // and semester
        public String Teacher;                  // the teacher of the course
        public _lesson lesson[];                // lessons that compose the course }

class _lesson extends AbstractEntry {
        static private URL DTDfile = new URL(“http://university.site/UnivCourse.dtd”);
        public String lessonname;               // the name of the lesson
        public Integer lessonnumber;            // the number of the lesson
        public String abstract;                 // the abstract of the lesson
        public _reading reading[];                      // suggested readings

class _reading extends AbstractEntry {
        static private URL DTDfile = new URL(“http://university.site/UnivCourse.dtd”);
        public String authors[];                // authors of the reading
        public String title;                    // title of the reading
        public String Book;                     // where it is published:
        public Integer volume;                  // volume
        public Integer number;                  // and number }
```

Figure 9. XMARS tuples corresponding to course, lesson, reading elements.

```
_lesson foundlesson;
...
_lesson templatelesson = new _lesson();
templatelesson.abstract = "*networks*"  // partially defined field
for(i=0; i< sites_of_the_federation.length;i++)          // for all the sites in the federation
{       go(sites_of_the_federation[i]);   // go to the current site in the list
        if ((foundlesson = S.read(templatelesson, new Transaction(null), NO_WAIT)) != null)
                                        // if a lesson containing "network" in the abstract is found
        go(home);                       // go back home
}
...
```

Figure 10. Code fragment of an agent searching for a specific lesson

```
_course updatecourse;
...
_course templatecourse = new _course();
templatecourse.teacher = "Jane Smith"  // defined field
for(i=0; i< sites_of_the_federation.length;i++)          // for all the sites in the federation
{       go(sites_of_the_federation[i]);   // go to the current site in the list
        if ((updatecourse = S.take(templatecourse, new Transaction(null), NO_WAIT)) != null)
        {       if(updatecourse.semester == 1) updatecourse.semester = 2;
                else                           updatecourse.semester = 1;
                S.write(updatecourse, new Transaction(null), 0);                }
}
go(home);                       // go back home
}
...
```

Figure 11. Code fragment of an agent in charge of updating semester information

Another simple example relates to information management. Let us consider that professor Jane Smith wants to update the information about the courses she is the teacher of. For example, let us suppose that she wants to exchange the semester during which her courses are held. To this end, she can deploy an agent that travels across the sites of the federation (or just across a few known sites storing information about her courses), extracts the information about a course and writes it again with the updated semester indication. Figure 11 shows a simple code fragment of agent performing this task. Of course, to perform its task, the agent launched by Jane Smith must be allowed to extract and write tuples from the XML documents containing information about her courses. However, it must not be allowed to extract and modify information about courses held by other teachers. To this end, since Access Control Lists in XMARS apply at the level of single document, security reasons require each course to be stored in a different XML document, thus forcing each site to adopt a specific file organisation for the XML dataspace. This clarifies why we consider the current Access Control Lists approach of XMARS inadequate.

In both the above examples, although very simple, XMARS programmability can provide advantages. With regard to the first example, a site can decide to associate a reaction to any read operation performed on its dataspace, with the goal of monitoring the activities on the dataspace and maintaining a log file. The code for this simple reaction is shown is figure 12 and it can be associated to any read event by writing the meta-level tuple (MonitorObj, read, null, null) in the meta-level tuple space (MonitorObj has to be an instance of the Monitor reaction class).

```
class Monitor implements Reactivity
{
public Entry reaction(XMARS s, Entry Fe, Operation Op, Identity Id)
{ SecurityRegister.add("read", Fe, Id);      // log the access
  return Fe }       // returns the matching tuple to the invoking agent
}
```

Figure 12. The Monitor reaction class

With regard to the second example, reactions can be associated to both take and write operation in order to preserve the consistency of the information in the dataspace, i.e., to guarantee that the information about a course is deleted from the dataspace only when the updated information is inserted. To this end, a reaction associated to the take operation must return the course tuple without deleting it from the space, and must keep record of the attempted take, for example via an *update record* that is a tuple stored in the space. The reaction associated to the writing of a new course tuple must check whether such record for the same course tuple exists, and delete the old version from the dataspace. Figure 13 shows the code of the classes that implement such reactions, which can be installed by writing the two meta-level tuples (PreventTakeObj, take, new _course(), null) and (RightUpdateObj, write, new _course(), null) in the meta-level tuple space. By specifying agent identities in the fourth field in the meta-level tuples, one can also discriminate which classes of agents must trigger the reactions and which must not. For instance, there may be the need of letting specific "administrator agents" perform take operations on course tuples without having to insert updated tuples. One could criticise that the collective behaviour of the PreventTake and RightUpdate reactions can be easily implemented by making use of the JavaSpaces' transaction mechanism. However, JavaSpaces' transactions require an agent-level identification of the problem. In XMARS, instead, the agent can disregard the problem, because the consistency of the XML dataspace is ensured from the internal, through a proper programming of its behaviour.

```
class PreventTake implements Reactivity
{
public Entry reaction(XMARS s, Entry Fe, Operation Op, Identity Id)
{
  UpdateTuple ut = new UpdateTuple();    // create the update record
  ut.coursename = ((_course)Fe).coursename;     // information about the deleted tuple
  ut.tuple = Fe;
  s.write(ut, new Transaction(null), 0);       // store the update record
   return Fe; }       // returns the matching tuple to the invoking agent without deleting it
}


class RightUpdate implements Reactivity
{
public Entry reaction(XMARS s, Entry Fe, Operation Op, Identity Id)
{ UpdateTuple ut = new UpdateTuple();
  ut.coursename = ((_course)Fe).coursename;
  UpdateTuple tupleupdated;
  if ((tupleupdated = s.take(ut, new Transaction(null), NO_WAIT)) != null)  // if there is an update
record for the same course
    s.take(ut.tuple, new Transaction(null), NO_WAIT);        // delete the old tuple
  return null;
}
```

Figure 13. The PreventTake and RightUpdate reaction classes

4.2 Inter-agent coordination

Coming to a more complex example, let us now suppose that the XML dataspaces, rather than being a simple repository of course material, becomes a general workplace for the management of the course activities. For instance, a specific XML dataspace can be exploited by professor Jane Smith for managing the periodic assignment for the students of her courses. She can publish the texts of the assignments in the dataspace, and her students can exploit the dataspace both to retrieve assignments and to make them available for evaluation. Each student can access the dataspace via a browser to select an assignment and, afterwards, to post the completed assignment (to be composed

in XML) in the dataspace. The teacher, by her side, can exploit a browser to access to the dataspace, read and evaluate the completed assignments.

Alternatively, each student can be provided with a personal user agent, in charge of accessing the dataspace, retrieving the assignment and, lately, storing the completed assignment in the dataspace. Analogously, the teacher can associate an agent with the dataspace, in charge of notifying her about the presence of a new completed assignment in the dataspace, and possibly capable of some preliminary controls about the correctness of the completed assignments. These controls may include checking that all of the parts of one assignment have been completed and/or verifying that no two assignments developed by two different students are the same. In addition, the teacher agent could verify whether assignments have been completed by the due deadline or not, in which case it can warn both the student and the teacher about the delay. Furthermore, the agent can control that assignments are submitted in the correct sequence by students: when an assignment is submitted before the preceding one, the agent can send a warning message to the student and should not notify to the teacher about the incorrect submission.

In the above sketched agent-mediated scenario, basic interactions between the student agents and the teacher one occur – accordingly to the Linda coordination model – via the insertion of XML elements/tuples representing assignments. The insertion of a completed assignment performed by one student's agent (via a write operation) triggers the teacher agent, which was waiting (via a blocking read operation) for these kinds of insertion events. However, the scenario also require these basic interactions to obey higher-order, application-specific, rules (specifically, workflow rules related to the order of assignment submissions), whose control is in charge of the teacher agent.

Let us now suppose that professor Jane Smith decides to change the rules related to the way student have to perform the assignments. For example, let us suppose that she starts publishing the specifications for the next assignment before the deadline for the completion of the preceding one, thus opening the possibility for students to complete assignments out of sequence. Unfortunately, this requires the teacher has to modify the code of its agent in according with the new workflow rules: once an assignment is submitted to the dataspace, the teacher's agent must no longer check that all previous assignments have been submitted too. The fact that the agent code has to be modified derives from an inappropriate design of the system component, which make the teacher agent in charge of controlling both the correctness of a submission in itself and the correctness of the submission in relation with the history of the submissions. In a more coordination-oriented perspective, the agent in not only in charge of handling the interaction events related to the insertion of a new assignment, but it also has to verify that this interaction obey specific coordination laws.

In the presence of a programmable dataspace model, a more clean solution can be sketched. The teacher's agent can be charged of controlling the correctness of the assignment content only, without worrying at all about whether students have to submit assignments in a given sequence or not. The latter problem, identifying a coordination law, can be charged to the interaction media itself, that is, the dataspace. In particular, in XMARS, the XML dataspace can be programmed in order to react to the insertion of a new assignment, and check whether the interaction event is respectful of the coordination laws. This achieves a clear separation of concerns and simplify agent design and development.

5. Related Work

While several mobile agent systems have been proposed in the last few years, and new ones keep on appearing, only a few of them focus on the definition of appropriate coordination models and/or on the adoption of XML as a language for agent interactions.

The *PageSpace* coordination architecture for interactive Web applications exploits Linda-like coordination [Cia98]: a multitude of tuple spaces in different execution environments can be used by agents to coordinate with each other; furthermore, agents can create private tuple spaces to

interact privately without affecting execution environments. However, unlike XMARS, PageSpace neither defines a programmable coordination model nor specifically focuses on the problem of standardising data representation toward interoperability. With regard to the latter problem, a foreign entity can be enabled to interoperate with a PageSpace application only by "agentifying" it, i.e., by associating it with a special-purpose PageSpace agent.

The LIME coordination model addresses in a unifying model both logical and physical mobility [PicMC99]. Each mobile entity – whether an agent or a physical device – has associated an Interface Tuple Space (ITS), which can be seen as a personal tuple space where to store and retrieve tuples. When mobile entities meet in the same physical place, their ITSs automatically merged, thus allowing Linda-like coordination between mobile entities via temporarily shared tuple spaces. LIME also provides for programmability of tuple spaces, although in a form more limited than the XMARS one: a mobile entity can program the behaviour of its own ITS only, in order to provide better control over the accesses to it. As a final note, being an abstract model rather than a concrete implementation, LIME currently neglects interoperability problems, such as data representation.

The *TuCSoN* model [OmiZ99], developed in the context of an affiliated research project, defines a programmable coordination model based on logic tuple spaces associated with the execution environments and to be used for the coordination of knowledge-oriented mobile agents. Logic tuple spaces define a Linda-like interface, while reactions are programmed as first-order logic tuples. The exploitation of logic programming in reaction complements the object-oriented XMARS reaction model and makes TuCSoN very well suited to the distributed management of large information systems. However, although the untyping of data of the logic model somehow facilitates interoperability (and it would also facilitate translation of logic tuples into an XML format), interoperability problems have to be solved in TuCSoN by explicitly programming "translator" reactions.

The *T Spaces* project at IBM [IBM98] defines Linda-like interaction spaces to be used as general-purpose information repositories for networked and mobile applications. In particular, T Spaces aims at providing a powerful and standard interface for accessing large amounts of information organised as a database. For this reason, T Spaces recognises the limits of the basic Linda model and integrates a peculiar form of programmability, by enabling new behaviours to be added to a tuple space, in terms of new admissible operations on a tuple space. This makes T Spaces less usable in the open Internet environment, since it requires application agents either to be aware of the operations available in a given tuple space or to somehow dynamically acquire this knowledge. With regard to interoperability, a project related to T Spaces and described in [AbrLC99] reports the only effort we have knowledge about for the integration of XML and Linda technologies. In particular, the goal of the project is to enable T Spaces to accept XML documents and store them as tuples in the tuple space. Incoming XML documents are processed to produce a DOM tree whose nodes are stored as T Spaces tuples, while tuples representing fragments of an XML file can be extracted to reconstruct the original XML document. This approach is somewhat dual of the XMARS one, where JavaSpaces tuples are translated for storing as fragments of an XML file and, viceversa, fragments of an XML file can be extracted and translated into JavaSpaces tuples. In our opinion, the XMARS approach has to be preferred. In fact, to preserve interoperability in a more efficient way, it is better to store XML files in their original format (as it may be required by other entities such as browsers, users, etc.), and produce tuples on-demand, when they are needed by some application agents.

The emergent importance of XML in the context of agent applications is stressed by the presence of several other proposals in the area, although not strictly related to tuple space coordination technologies. In [NakY98], XML is used to represent agent interaction protocols: by exchanging XML messages accordingly to a specific schema, communication sessions between agents can be easily established and the order of the exchanged messages easily controlled. An XML framework for agent-mediated E-commerce is described in [GluTM99]: the framework provides an extensible library of generic XML document templates, to be used to facilitate the

exchange of information among electronic suppliers, resellers and customers, and build up an open and flexible e-commerce system. Other approaches go further, and exploit XML as a programming language for agents. In [Xage98], data and status of agents are represented in XML format, while their code can be written in a scripting language (such as JavaScript) and enclosed in specific XML tags. In [LanHO99], the agent code itself can be written in XML, by expressing control instructions in terms of XML tags (e.g., <if>condition</if>).

6. Conclusions and Future Work

The presented XMARS architecture, by exploiting the power and the flexibility of a programmable Linda-like coordination model in the context of XML dataspaces, may provide several advantages in Internet applications based on mobile agents. In fact, while programmable Linda-like coordination suits the mobility of the application components and the openness of the Internet scenario, the emergent XML standard for Internet data representation may guarantee a high-degree of interoperability between heterogeneous components.

However, apart from the need of updating portions of the current implementation (as discussed section 3), there are still several issues to be solved to make the XMARS architecture practically usable. These issues mainly relates to the lack of some XML specifications that are instead necessary for any effective management (and consequently for a tuple-based management) of XML documents [Tau99]. First of all, the *XML schemas* specification will permit to better specify data types in XML documents than the current version; this specification, together with the integration of XML namespaces in the current implementation, will be of great help in translating XML fields into Java objects (and viceversa), and in improving the effectiveness of the pattern matching mechanism. Second, the *XML fragments* specification will precisely define how fragments of an XML document can be extracted and inserted in a harmless way (i.e., preserving the validity of the XML document itself), thus meeting the need of handling single elements inside a possibly long document and, in the context of XMARS, enabling the system to extract/insert tuples representing a fragment of a large document in a consistent way. Strictly related, the *XLink* and *XPointer* specifications, which will rule the connections between different (parts of) XML documents, will possibly lead to richer and more complex tuple types.

7. References

[AbrLC99] J. Abraham, H. Le, C. Cedro, "XML Repository In T Spaces & UIA Event Notification Application", http://www.cse.scu.edu/projects/1998-99/project19/, 1999.

[AhuCG86] S. Ahuja, N. Carriero, D. Gelernter, "Linda and Friends", *IEEE Computer*, Vol. 19, No. 8, pp. 26-34, August 1986.

[Auth98] The Authors, "Reactive Tuple Spaces for Mobile Agent Coordination", 2nd International Workshop on Mobile Agents, LNCS, No. 1477, pp. 237-248, Springer-Verlag (D), Sept. 1998.

[Auth99] The Authors, "Coordination Models for Internet Applications based on Mobile Agents", IEEE Computer Magazine, 1999, to appear.

[CDF97] Channel Definition Format, http://www.w3.org/TR/NOTE-CDFsubmit.html, 1997

[Cia98] P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, A. Knoche, "Coordinating Multi-Agents Applications on the WWW: a Reference Architecture", IEEE Transactions on Software Engineering, Vol. 24, No. 8, pp. 362-375, May 1998.

[CiaVM99] P. Ciancarini, F. Vitali, C. Mascolo, "Managing complex documents over the WWW: a case study for XML", IEEE Transactions on Knowledge and Data Engineering, Vol. 11, No. 4, 1999, to appear.

[DenNO98] E. Denti, A. Natali, A. Omicini, "On the Expressive Power of a Language for Programmable Coordination Media", 1998 ACM Symposium on Applied Computing, Atlanta (G), Feb. 1998.

[FugPV98] A. Fuggetta, G. Picco, G. Vigna, “Understanding Code Mobility”, IEEE Transactions on Software Engineering, Vol. 24, No. 5, pp. 352-361, May 1998.

[GelC92] D. Gelernter, N. Carriero, “Coordination Languages and Their Significance”, Communications of the ACM, Vol. 35, No. 2, pp. 96-107, February 1992.

[GluTM99] R. J. Glushko, J. M. Tenenbaum, B. Meltzer, "An XML-Framework for Agent-based E-commerce", Communications of the ACM, Vol.42, No.3, pp.106-114, March 1999.

[IBM98] T Spaces Home Page, IBM, http://www.almaden.ibm.com/Tspaces, 1998.

[JS98] Sun Microsystems, *JavaSpaces Technology*, http://java.sun.com/products/javaspaces/, 1998.

[KarT98] N. M. Karnik, A. R. Tripathi, “Design Issues in Mobile-Agent Programming Systems”, IEEE Concurrency, Vol. 6, No. 3, pp. 52-61, July-September 1998.

[KhaR97] R. Khare, A. Rifkin, “Capturing the State of Distributed Systems with XML”, The World Wide Web Journal, Vol.2, No.4, pp.207-218, Fall 1997.

[LanHO99] D. B. Lange, T. Hill, M. Oshima, "A New Internet Agent Scripting Language Using XML", AAAI-99 Workshop on AI in Electronic Commerce, July 1999

[NakY98] Y. Nakamura, G. Yamamoto, “A Framework for representing Agent Interaction Protocols based on XML”, IBM Research, Tokyo Research Laboratory, Japan, September 1998.

[OMG97] OMG, *CORBA 2.1 specifications*, http://www.omg.org, 1997.

[OmiZ99] A. Omicini, F. Zambonelli, “Coordination for Internet Application Development”, Journal of Autonomous Agents and Multi-agent Systems, Vol. 2, No. 3, pp. 251-269, Sept. 1999.

[PapA98] G. A. Papadopoulos and F. Arbab, “Coordination Models and Languages”, Advances in Computers, Vol. 46, pp. 329-400, Academic Press, Aug. 1998.

[PicMC99] G. P. Picco, Amy L. Murphy, G. Catalin Roman, "Lime: Linda Meets Mobility", International Conference on Software Engineering, Los Angeles (CA), May 1999.

[Tau99] J. Tauber, “XML after 1.0: You Ain’t Seen Nothin’ Yet”, IEEE Internet Computing, Vol. 3, No. 3, pp. 100-102, May-June 1999

[W3Ca] The World Wide Web Consortium, eXtensible Markup Language pages, http://www.w3.org/XML

[W3Cb] The World Wide Web Consortium, eXtensible Stylesheet Language pages, http://www.w3.org/Style/XSL

[WonPM99] D. Wong, N. Paciorecк, D. Moore, "Java-based Mobile Agents", Communications of the ACM, Vol.42, No.3, pp.92-102, March 1999.

[Xage98] The XML Agents (XML-A) Initiative, http://www.jxml.com/xmlagent.html, 1998.