

MEnSA Project

Methodologies for the Engineering of complex
Software systems: Agent-based approach

Deliverable 3.1

Ambra Molesini, Andrea Omicini

15th October 2007

Date	30th September, 2007
Product Type	Deliverable
Pages	
Responsible Unit	UNIBO
People involved	Ambra Molesini, Andrea Omicini, Giacomo Cabri, Raian Ali
Contact Author	Ambra Molesini

Abstract

This deliverable presents the state of the art in the field of agent-oriented infrastructures and the role of the infrastructures in the engineering of the complex systems. Our objective is to understand and clarify the concepts, relationships and dynamics used in each infrastructure. We present the static and dynamic features of the infrastructures by means of two different models: the static model depicts the meta-models of the infrastructures while the dynamic model represent what happens at run-time.

Contents

1	Introduction	2
1.1	Infrastructures for MAS	3
1.1.1	Enabling vs. Governing Infrastructures	4
1.2	Infrastructures' Models	6
1.2.1	Static Model	7
1.2.2	Dynamic Model	8
2	Infrastructures and Software Engineering	11
2.1	Infrastructures and Software Engineering	11
2.1.1	Infrastructure Selection	12
2.1.2	A Sketch of the State of the Art	14
2.2	Infrastructures and AOSE	14
2.3	Coordination, Organisation and Security in the same Engineering Context	17
3	Agent Oriented Infrastructures	19
3.1	JADE	19
3.1.1	Static Model	19
3.1.2	Dynamic Model	22
3.2	TuCSon	24
3.2.1	Static Model	24
3.2.2	Dynamic Model	26
3.3	MARS	28
3.3.1	Static Model	28
3.3.2	Dynamic Model	30
3.4	TOTA	32
3.4.1	Static Model	32
3.4.2	Dynamic Model	33
3.5	RoleX	35
3.5.1	Static Model	35
3.5.2	Dynamic Model	37
3.6	CARTAGO	39
3.6.1	Static Model	40
3.6.2	Dynamic Model	41
4	Conclusions	44
	Bibliography	45

1

Introduction

Today, infrastructure is a fundamental notion for complex systems in general, not only in computer science and engineering, but also in the context of organisational, political, economical and social sciences. In its most general acceptation, an infrastructure is defined as [35]:

Merriam-Webster – (1) the underlying foundation or basic framework (as of a system or organisation) (2) the permanent installations required for military purposes; (3) the system of public works of a country, state, or region; also: the resources (as personnel, buildings, or equipment) required for an activity;

Cambridge – (4) the basic systems and services, such as transport and power supplies, that a country or organisation uses in order to work effectively;

The American Heritage – (5) the basic facilities, services, and installations needed for the functioning of a community or society, such as transportation and communications systems, water and power lines, and public institutions including schools, post offices, and prisons.

Every definition underlines the role of infrastructure as (part of) the environment that provides basic resources and critical services to complex systems (such as organisations, communities, societies, countries) living on top of it. In particular, definition (2) remarks the fact that an infrastructure is a persistent entity: once installed, an infrastructure typically survives the many systems it supports. Also, definitions (4) and (5) remark the key role of infrastructures: their services typically cover critical system issues, and provide features that individual system components could not afford to provide or obtain elsewhere. In the context of MAS, infrastructure obviously plays a key role, given the potential complexity of both the system components (agents) and the component interplay (agent societies). Gasser [14] defines an infrastructure as:

“a technical and social substrate that stabilises and rapidly enables instrumental (domain-centric, intentional) activity in a given domain... (solving) typical, costly, commonly accepted community (technical) problems in a systematic and appropriate ways”

Here, it is important to emphasise the notion of infrastructure as a social, enabling support for providing MAS with cheap and systematic solutions to common problems. Another interesting definition is provided by Sycara et al. [50]:

“Agents in a MAS are expected to coordinate by exchanging services and information, to be able to follow complex negotiation protocols, to agree on commitments and to perform other socially complex operations. We define the infrastructure of a MAS as the set of services, conventions, and knowledge that support such complex interactions.”

The stress is here on the support of complex agent (social) interplay, which is expressed in terms of services, convention and knowledge.

1.1 Infrastructures for MAS

In a more abstract acceptance than the ones above, the main role of infrastructures in MAS is to model and shape the *agent environment*, from the two points of view (i) of the agents living in the MAS; and (ii) of MAS designers. From the inner viewpoint of an individual agent, the infrastructure typically provides the means to deal with the agent environment: to perceive and affect its state and dynamics (in general), to access resources and services, to obtain and store information, to interact with other agents (in particular). Typically, a suitably expressive and well-engineered infrastructure allows agents to represent their environment only through the runtime abstractions provided by the infrastructure, and to modify the agent environment according to the agent's needs and goals through infrastructure services. From the external viewpoint of a human designer, MAS are typically open systems, both in terms of the unpredictability of their environment (due to components and interactions not under the control of MAS designers), and of the dynamism of both MAS structures (e.g., the set of agents in a MAS) and MAS processes as well (e.g., the coordination activities within a MAS). Infrastructures are then the suitable place for designers to embed elements of control of MAS despite their inherent openness: such control can be exerted by means of runtime abstractions provided by the infrastructure that can embody and

enforce interaction constraints, coordination laws and social norms. Even more, once they are suitably described and made accessible to agents, the same runtime abstractions can be exploited by intelligent agents in order to represent coercive structures of a MAS, and to act upon its global behaviour by introducing and/or modifying constraints, laws and norms [36].

Infrastructures play then a key role in the engineering of MAS, too. This is quite obvious when considering the last stages of the engineering process, that is, the development and deployment of MAS. Nevertheless this also holds when taking the early stages into account, that is, the modelling and design of MAS: the abstractions provided by the infrastructure could be good candidates to be adopted and exploited in the design of MAS structures and activities, which are then to be engineered on top of such abstractions. So, runtime abstractions should be flexible enough to support the engineering of heterogeneous systems, and – at the same time – effective in minimising the gap between the design and development / deployment / runtime of systems.

In this context, the tools provided by an infrastructure are fundamental to enable the manipulation of the abstractions through all the engineering stages, in particular at runtime. The definition of the engineering tools is a primary issue, that should be necessarily inspired and driven by the model embodied by the MAS infrastructure itself [11]. In the end, MAS infrastructures and tools play an essential engineering role by keeping abstractions alive through the whole engineering process, thus enabling software engineers to first design and then observe and act on MAS structures and processes at runtime, working upon abstractions adopted and exploited for the design of a MAS. This feature is particularly important to support forms of online engineering, i.e., the capability of supporting system design / development / evolution while the systems are running – a particularly relevant feature in the context of MAS, given their intrinsic complexity and openness.

1.1.1 Enabling vs. Governing Infrastructures

Infrastructures are useful to encapsulate and support critical features and properties of MAS; these properties typically concern the interaction dimension. For this extent, current MAS infrastructures can be considered enabling infrastructures, since they provide abstractions that basically enable agent interaction at different levels: from communication to interoperability, to basic interaction services. This is apparent when considering the abstract architecture of one of the most important infrastructure currently adopted for MAS development and deployment: JADE [5]. There, in fact, services like agent communication, inter-operation, security, naming, location, etc., are necessary preconditions that make it possible for agents to live, coexist and interact

within a MAS. Enabling infrastructures, then, basically define the nature of the agent interaction space within a MAS. However, the increasing complexity and articulation of MAS for today’s application scenarios call for a most effective engineering support from infrastructure, beyond the mere enabling of agent interaction. A well known example are Electronic Institutions [28]: the social and normative capabilities required to infrastructures supporting eInstitutions go far beyond the services provided by general purpose MAS enabling infrastructures, and cannot be straightforwardly engineered on top of it. Another example comes from team-oriented coordination: in order to be independent of the specific agent model, the TEAMCORE [52, 53] approach introduces the PROXY abstraction, an infrastructure component provided to agents for managing automatically all coordination dependencies with respect to the teams that agents belong to [51]. Similar team-oriented capability has been added to RETSINA by enhancing its Individual Agent Architecture [17]: in this way, contrary to the TEAMCORE approach, no real infrastructure support is provided from the infrastructure to team-oriented coordination, since the team-oriented capability is obtained by relying on augmented capabilities of the individual agents. In the end, current general purpose MAS infrastructures typically lack of suitable abstractions to govern agent interaction. This seems instead a fundamental feature for enabling the specification and enactment of social norms, but also – more generally – for defining and executing social activities, such as agent coordination. In other words, complex system engineering calls for governing infrastructures, providing flexible and robust abstractions to model and shape the agent interaction space, in accordance with the social and normative objectives of systems.

Governing infrastructures become the natural loci where to embody a conceptual framework that uniformly accounts for organisation, coordination and security of MAS altogether [37]. From the organisational point of view, infrastructures are to provide explicit abstractions for modelling the structure of an organisation and its rules – e.g., using the notion of role and related permissions to access to resources. This is the case, for instance, of the information system infrastructure that support the RBAC model [47], which is attracting attention also in the context of MAS. From the coordination point of view, infrastructure support can be described effectively by adopting the notion of coordination as a service [34, 55]: according to this vision, the infrastructure itself is the provider of runtime (coordination) abstractions designed for specifically supporting the specification, execution and maintenance of MAS social activities. These abstractions become a fundamental tool to face the engineering complexity of coordination in MAS: both from the designer’s and the agents’ point of view, the coordination bur-

den is distributed between agents and the specialised services provided by the infrastructure. The coordination abstraction is promoted to a first class entity in coordinated systems, amenable of an explicit characterisation in terms of an interactive abstraction, with the goal of supporting the design of coordination models down to the deployment of coordination infrastructures. Coordination abstraction is the conceptual locus where coordination takes place. The expressiveness and flexibility of coordination abstractions strongly influence the engineering of social activities, and, consequently, the complexity of the solutions adopted for the challenging application scenarios. Since they are part of the infrastructure, these coordination abstractions are typically expected to be robust and reliable, and specifically designed to support a critical activity as coordination is.

Two observations are worthwhile here. Firstly, the evolution from enabling to governing infrastructures can be devised also in other computer science fields, characterised as well by complex organisations and collaboration activities: Computer Supported Cooperative Work (CSCW) and Workflow Management are relevant examples. Especially in the CSCW context the need for suitable infrastructure support for coordination has already emerged as a fundamental issue. Schmidt and Simone [48], for instance, identify basic properties that coordination abstractions provided by an infrastructure should feature. Secondly, the approach of coordination as a service has also a deep impact on AOSE methodologies, since coordination abstractions – as they embody the social aspect of MAS – are meant to become explicitly subject of all the engineering stages, as it happens in **SODA** methodology [2, 26, 27].

1.2 Infrastructures' Models

At the best of our knowledge, in the infrastructures' research field there is not a standard language for modeling and representing an infrastructure. Taking inspiration from the traditional languages used in the Software Engineering, we use UML [30] as a language for modelling the infrastructures presented in the next Chapter 3. In particular we use the UML's class diagram for representing the static model (Subsection 1.2.1) and the UML's sequence diagram for representing the dynamic model (Subsection 1.2.2).

We have chosen UML as notation after that a careful evaluation of other specification languages such as OPM [12]. UML is well-known and a standard in the software engineering mainstream, even if it does not allow the engineer to express in a satisfactory way all the complex interactions that occur in the infrastructures.

The use of UML should not be misunderstood. We use UML only as a notation language for representing a high-level description of the infrastructures. This does not imply that our models represent the “real” infrastructure’s structure, and also does not imply an object-oriented implementation of the infrastructure anyhow. The UML language presents limitations when used as a metamodeling tool, some of them – relative to class diagram – are highlighted in [24].

In the reminder of this section we briefly present the static and dynamic models that we adopt in Chapter 3.

1.2.1 Static Model

The static model describes the structure of a system in terms of abstract elements and the relationships among them. As a tool for representing the infrastructure’s static model we have chosen the meta-model technique, which is already successfully adopted in the context of the methodologies [6, 54].

As for the methodologies, all infrastructures introduce some basic “run-time abstractions” (agents, organisations, resources, . . .) in order to support deployment and execution of systems. So, it is not so strange to represent the relationships between such abstractions by means of a *meta-model*. A meta-model is a model of concepts that can be used to describe actual systems—an infrastructure in our specific case. It enables checking and verifying the completeness and expressiveness of an infrastructure by understanding its deep semantics, as well as the relationships among concepts.

Thus, the infrastructure’s meta-model becomes the key tool to compare an infrastructure with each other, and to check the consistency of an infrastructure when planning extensions or modifications. Meta-models are also an important guide for integrating different infrastructures avoiding several errors, such as assuming differences of concern when none exists, or assuming similarity of concern because of a common use of terms despite a different semantics. In addition, infrastructure’s meta-models could be used for associating each methodology’s concept to some suitable infrastructural abstraction(s): studying and comparing methodologies with infrastructures in terms of meta-models makes it possible to provide guidelines for mapping the design model of a methodology onto its implementation [25].

Among the possible choices we have decided to adopt the UML as a notation language for representing the meta-model. In particular we use the class diagram. A class diagram is a type of static structure diagram that describes the structure of a system by showing the system’s classes, their attributes, and the relationships between the classes.

Obviously the semantics we associate to the UML's elements is quite different from the "traditional" semantics used in the object-oriented context. For example in the class diagram the "class elements" are not to be intended as the traditional object-oriented classes but simply as abstract entities supported by infrastructures.

1.2.2 Dynamic Model

The dynamic model describes and models the behaviour of the system over time. For the representation of this model we evaluated different alternatives such as OPM [12] and UML.

OPM's basic principle is that structure and behaviour in a system are so intertwined that effectively separating them is extremely harmful, if not impossible: using stand-alone processes, one can model a behaviour that involves several object classes and is integrated into the system structure. OPM could be a good choice for representing the dynamic model, however OPM is not well-known as UML, so as a tool for representing the infrastructure's dynamic model we have chosen the UML sequence diagrams.

A sequence diagram shows, as parallel vertical lines, different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur. This allows the specification of simple runtime scenarios in a graphical manner.

As for the class diagram, the semantics we associate to the language's elements is quite different from the "traditional" UML semantics. In particular, the only "interaction mechanism" supported by sequence diagram is message passing. In fact a sequence diagram provides a sequential map of messages passed between objects over time. So in the sequence diagrams we show in the Chapter 3, the interactions among infrastructure's entities are represented only by means of a message passing technique. This is obviously an abstraction over the real mechanisms adopted by each infrastructure. Each infrastructure, in fact, support its own interaction mechanism: some infrastructures use message passing (like Jade), while others use different forms of mediated interaction (like **TuCSon**, TOTA,...). In a similar way, even if the interaction among abstract entities is illustrated with a synchronous semantics in the sequence diagrams, the interaction in the infrastructures does not necessarily occur in a synchronous way.

Another limitation of the sequence diagram is the difficulty to express "alternative paths" in the same diagram in order to have a global view of the interactions. For example, let us suppose to represent the interactions among three different entities A, B, C (Figure 1.1). The interaction initiator is the entity A that sends a message to B. B executes an internal elaboration

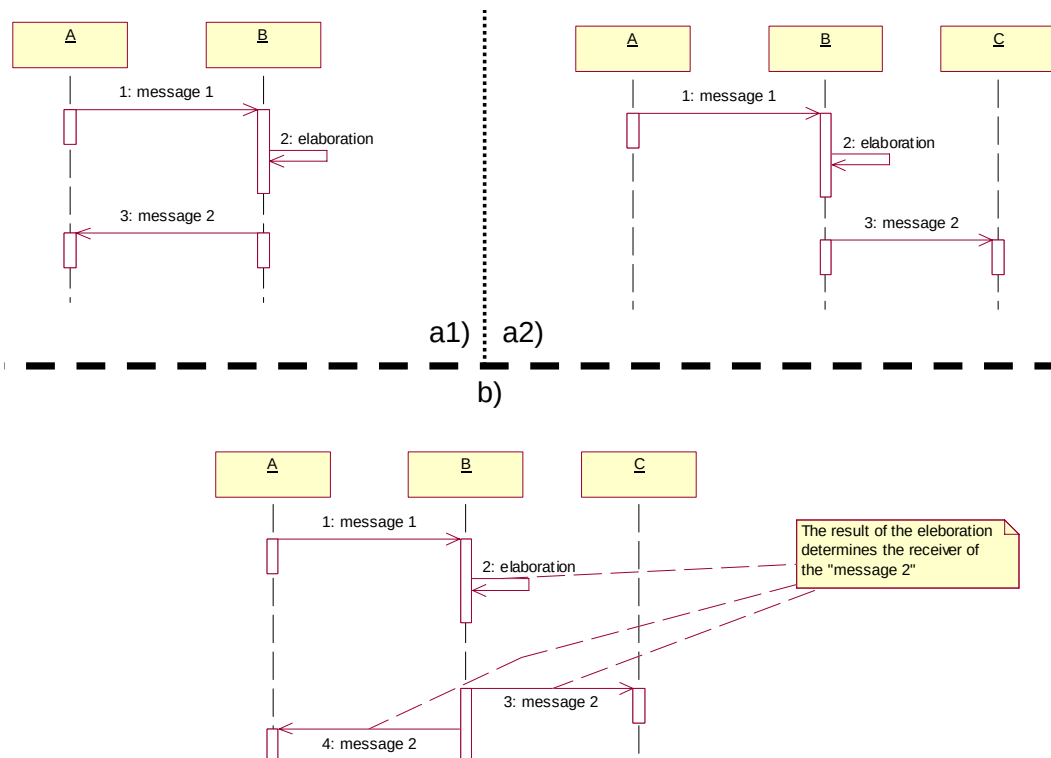


Figure 1.1: Problem in Sequence Diagram: two different diagram express the interaction (top), and the interaction in the same diagram (bottom)

and the result of this elaboration determines the receiver of the message that B will send. This is an example of “alternative paths”: the interactions among entities depend on the result of internal elaboration. There are two possible ways to represent this kind of interaction, as illustrated in Figure 1.1. The first solution – top of the figure – represents each alternative path in a different diagram. Instead, the second solution – bottom of the figure – depicts the different paths in the same diagram with “explanatory notes” that explain the alternatives.

Even though both solutions can capture alternative path, each of them presents drawbacks. The first solution leads to an easier-to-read dynamic model than the second. However, the first solution also contains too many diagrams, so the general overview of the system behaviour is difficult to obtain. Instead, the second solution leads to a more compact model, which is however less readable, in particular when there are a lot of alternative paths.

For this work we have chosen the first solution both for the better readability and for a possible future integration of all the different alternative

diagrams relative to the same interaction in a unique Interaction Overview diagram. This diagram is supported in the UML version 2.0, but at the time of writing it is not so known as the sequence diagram and it is not so supported by the different UML CASE tools.

2

Infrastructures and Software Engineering

2.1 Infrastructures and Software Engineering

As mentioned above, an infrastructure – or *middleware*, as it is often called in traditional Software Engineering (SE) field – is a technical and social substrate that stabilises and rapidly enables instrumental (domain-centric, intentional) activity in a given domain [14]. Said another way, infrastructure solves typical, costly, commonly-accepted community problems in systematic and appropriate ways. So, infrastructure allows much greater community attention to unique, domain-specific activities.

For example, the construction of a distributed systems is considerably more difficult than building a centralised or client/server system. This is because there are multiple points of failure in a distributed system, heterogeneous components need to communicate with each other through a network, which complicates communication and opens the door for security attacks. Infrastructure has been devised in order to conceal these difficulties from application engineers as much as possible [10]. Infrastructure resolves heterogeneity, and facilitates communication and coordination of distributed components.

Existing infrastructure products enable software engineers to build systems that are distributed across a local-area network, and they offer pre-built services that support, for example, off-the-shelf distributed transaction processing, security, and directory and naming services. The infrastructures also provide specialised components that let developers integrate many different legacy systems and design and deploy new business processes that integrate multiple distributed applications.

From the software engineering view point the infrastructures present two key challenges illustrated in the following: the infrastructure selection (Subsection 2.1.1) and the impact of the infrastructures in the engineering process.

At the best of our knowledge, no work in literature explicitly deal with the identification of how the infrastructure impacts in the software engineering process. This appears very strange because of the presence of a specific infrastructure could influence the engineering process. For example, an engineer could design or could not design specific functions according to the services / functionality provided by the infrastructure. This is specially true in the context of on-line engineering where the infrastructure's abstractions have a great impact in the engineering process.

2.1.1 Infrastructure Selection

Two widely used but very different infrastructure styles (see Subsection 2.1.2) are object-oriented and message-oriented. Within each of these styles, there are multiple products to choose from. Moreover, any of these products may be used alone or in combination with other products. Thus the problem of infrastructure selection is increasingly important in the engineering of enterprise software systems [19].

Infrastructure selection is important for a number of reasons. It is an essential part of the way in which distributed systems get built, both by new development and by integration of existing applications and services. Moreover, infrastructure is a key enabling technology: it provides services, supports application functions and features, separates concerns, and integrates components [19]. In these roles, infrastructure interacts with and may impact on many other kinds of technologies, such as database systems, workflow engines, web servers, and applications. It further affects system architecture and development processes.

Infrastructure selection is also challenging, for these same reasons. Infrastructure selection can be an involved software (and systems) engineering process in its own right, with all the technological, organisational, economic, and political aspects that this may imply. Because of the central position and critical function of infrastructure, if it is selected or applied inappropriately, it can become a key disabling technology. Integration projects may present more or fewer challenges for infrastructure selection, depending on the diversity of systems and applications to be integrated. For new development, the constraints on infrastructure may be initially less restrictive, but infrastructure selection must be closely integrated with the design of other elements of the system, so selection can still be complex. In addition, the role of infrastructure selection must be defined with respect to the rest of the software life cycle. Infrastructure selection should generally be based on system requirements specifications (that account for both technological and organisational factors), but infrastructure selection should not be wholly de-

terminated by those specifications. Depending on circumstances, infrastructure selection may occur prior to, along with, or subsequent to system architectural specifications. Consequently, infrastructure selection may constrain, or be constrained by, architectural design.

Despite the potential variety of middleware selection processes, a number of guideline or recommendations regarding infrastructure selection processes can be identified [19]:

- selection may be made based on a number of approaches to comparing and evaluating infrastructure technologies;
- the frameworks, taxonomies, and criteria that are used for comparing and evaluating infrastructure must reflect enterprise-specific and project-relevant conditions;
- standards can be useful for enhancing communication among stakeholders as well as facilitating technical integration;
- the selection process should allow for the adoption of multiple alternative (or complementary) technologies in the solution to any particular problem.

Three other important factors should be considered in the infrastructure selection [39]:

- *Programming paradigm.* Adopting a new technology changes the concepts, the techniques and the programming styles used to create software and to reason about it(*paradigm shift*).
- *Development Process.* Adopting a new technology affects the actual software development activities and how they are related.
- *Economical Environment.* Adopting a new technology can advantage some stakeholders more than others, and a market situation can turn out to be more or less favorable to a new technology.

An example of an infrastructure selection process can be found in [20]. The authors present the i-Mate process that offers three tools to help organisations more effectively engineer Internet-based applications that require infrastructure technology. i-Mate provides a defined process for gathering, ranking, and weighting application requirements that the infrastructure must meet.

2.1.2 A Sketch of the State of the Art

Current middleware technologies fall into four broad categories: Corba-based technology, Java-based technology, messaging and integration technologies, proprietary technologies [20].

Corba-based technologies [29] enables transparent, synchronous inter-object communication across different processes, written in different languages, running on different hosts. These technologies hide the complexities of low-level networking details from the application programmer and provide easy integration across heterogeneous computing platforms. Further, IIOP (Internet Inter-ORB Protocol) enables integration of systems built on different Corba products.

Java-based technologies have evolved through Sun's Java 2 Enterprise Edition specification (J2EE) [49]. This defines the Enterprise Java Beans programming model, which is built on the Java Remote Method Invocation (RMI) communication protocol. It enables communication between different Java components across different machines and simplifies many distributed-programming issues by incorporating transaction services, component life-cycle, integration and persistence services, and security.

Messaging and integration technologies include message-oriented middleware such as IBM's Websphere MQ family [18]. These widely deployed messaging products provide asynchronous communication models for autonomous systems to exchange messages in both point-to-point and multicast modes. Add-on services such as message transformation and translation engines simplify business integration by unifying message formats and offering event-triggering services.

Proprietary technologies include for example Microsoft .Net platform [23], which provides component-based technology for deploying applications on the Windows platform. .Net includes class libraries that give components access to base-level middleware services. All .Net compilers then generate a common intermediate code representation that is translated to machine code as the application executes. .Net also includes extensive support for Web services technologies, which provide a nonproprietary access mechanism for .Net applications.

2.2 Infrastructures and AOSE

The availability of infrastructures has a considerable impact on the process of MAS engineering, and therefore should play a significant role within agent-oriented methodologies. As already mentioned, infrastructures impact on

both the final stages of the engineering process (development and deployment) as well as on the analysis and design stages, by means of the abstractions provided by the infrastructure model to represent the environment and to support coordination and organisation [35].

Infrastructure should provide flexible and robust abstractions to model and shape the agent interaction space, in accordance with the social and normative objectives of systems. As illustrated for instance by the **SODA** methodology [2], coordination abstractions provided by a MAS infrastructure should represent the runtime embodiment of the same analysis / design abstractions used since the early stages of the MAS engineering process. Keeping abstractions alive along the whole engineering process is in fact essential to support advanced practices like *online engineering*: there is no viable way to evolve a system online when the design abstractions are no longer in place in the runtime. Here, then, infrastructures should play a key role: they should provide MAS engineers with the same abstractions used in the analysis / design phases, as well as the tools to for their off-line / online development, deployment, monitoring and debugging.

Infrastructures also represent an effective approach to the general issue of formalisability of complex systems, which may come from either pragmatical or theoretical problems [38]. By their very nature, infrastructures intrinsically encapsulate key portions of systems –often in charge of the critical system behaviour. As a result, providing well-specified infrastructures promotes the discovery and proof of critical system properties. Most notably, a system property can be assessed at design time through the formal definition of some design abstraction. Then, by ensuring compliance of the corresponding run-time abstraction provided by the infrastructure, such a property can be enforced at execution time and be automatically verified for any system based on the infrastructure.

Infrastructures, being the concrete realisation of a conceptual model with software artefacts, can help with technology transfer in all the three dimensions described above (the last three points in Subsection 2.1.1). In short [39],

- the paradigm shift or adaptation is eased by a well-defined set of APIs,
- infrastructure can effectively cooperate with tools to shape the development process,
- and a good quality, freely available infrastructure can rapidly increase the adoption of a new technology, thereby creating market opportunities for services and applications based on that technology

While the above applies to infrastructure in general, the agent-oriented infrastructure has some more peculiar features of its own. In the case of agent technology, the pivotal concept of *situated agent* brings about some interesting consequences when considered in the context of heterogeneous software integration. In complex applications, where a high degree of system integration is needed, a trend towards seamlessly situated agents can be observed; that is, a multi-agent system can be embedded in an application and work together with other sub-systems adopting different component models. In principle, this is in contrast with the *wrapper agent* approach, where every non-agent software entity or subsystem has to be hidden behind an agent. The authors observe the trend towards seamlessness during their active experience with agent technology in the past few years, and think this is the goal that agent infrastructure has to pursue in order to effectively achieve the mainstreaming of agent technology.

From the perspective of a software agent, the outcome is that most interactions with non-agent software are modelled as interactions with the physical environment [39]. So, the balance between social and physical environment critically depends on the amount of agent wrapping performed on non-agent components. In a highly wrapped solution, such as the one specified in the FIPA Agent-Software Integration document [13], nearly all interactions are likely to occur through ACL. If agents, instead, directly interact with most non-agent software entities, ACL usage will be used for interactions about those entities and not with the entities.

Language is about representation, so while it is not possible to physically act on a language-level entity, we often have the need to talk about physical domain entities. This means that the actions and events that describe the physical situatedness of a software agent must also have representations at the language level. The most common way to achieve this is by combining ACL and domain ontologies – this is the approach followed by FIPA specifications, too. The greater balance between the social and the physical environment of an agent, spurred by the trend towards seamlessly situated agents, puts forth the role played by the dimension of *interaction*.

There, in fact, complexity is no longer simply bound to the granularity of the individual system's components, but rather to the many different ways in which components relates to and interact with each others, and with the overall system, too. Managing and governing the dimension of interaction is exactly the goal of *coordination*, which has emerged as a fundamental research areas in many several fields – from software engineering to artificial intelligence, from social theory to economics, from organisational theory to biology. The coordination is perceived as issue to be the main challenge to address and solve in order to actually obtain seamlessly situated MASs; the

complex spectrum that spans from purely linguistic interaction to lower level physical-like perception must be covered and effectively exploited to really connect agents with their social and physical environment.

2.3 Coordination, Organisation and Security in the same Engineering Context

In the context of MAS, organisation and coordination are strictly related and interdependent issues, and so MAS coordination infrastructures have a fundamental engineering role also in MAS organisation [36]. Generally speaking, organisation mainly deals with the structure and the long-term relationships between the components of a system, while coordination mainly concerns the processes and the dynamic interactions between the components of a system – often related to roles that usually frame agents in the structure / pattern of system organisation. In any case, both organisation and coordination concern and affect the way in which agents interact with each other, so that conceiving and representing them in the same framework is likely to provide several advantages. Conceptual economy is obviously the first benefit: for instance, the notion of role, usually introduced by organisational models, typically constrains agent actions, which is one of the corner-stones of coordination. Also, a common framework is the most obvious way to consistently support adaptation and evolution of organisation and coordination within an agent society: for instance, by managing explicitly the dependencies between the changes in the organisational settings (such as removal of a role, or changes in its capabilities in terms of interaction protocols) and the related effects on coordination activities. Even more, there are system aspects that can be modelled and engineered in their complex articulation only by considering organisation and coordination settings at the same time: security and electronic institutions are well-known examples. In particular, the multiple aspects related to the security issue in MAS can be tackled in a coherent and satisfactory framework only by covering the whole spectrum that ranges from organisation – with issues related to system structures and relations among the components – to coordination with issues related to collective processes. Facing security modelling and engineering within this range increases system conceptual integrity, by promoting the reuse of abstractions such as roles, permissions, and societies – which have already proved to be effective in the context of organisation and coordination – in order to enforce complex and dynamic security policies.

Even though the need for run-time liveness of design abstractions sup-

ported by the MAS infrastructure follows from basic system engineering considerations, it has an impact on the engineering of intelligent systems [31]. When dealing with MAS organisation abstractions, their liveness allows in principle to dynamically inspect and, possibly, change or adapt it. This is obviously useful for promoting human activities over systems such as monitoring and incremental evolution: however, when dealing with intelligent systems, the liveness of (organisation/coordination) abstractions is particularly relevant since the properties they embody can be in principle made available not only to humans, but also to intelligent agents. This clearly promotes self-reconfiguration and self-adaptation of intelligent systems: in fact, once an intelligent agent is enabled to inspect the social structure, and allowed to change it, it may reason about the organisation, make inferences, and possibly plan its evolution, for instance to fix some undesired behaviour, or to adapt to environmental changes [36].

Summing up, it is both possible and useful to conceive a MAS infrastructure that supports the modelling and enactment of organisation aspects in synergy with the coordination ones, by keeping the abstractions alive throughout the whole engineering process: that is, by providing MAS engineers with design abstractions also suitable for organisations (such as the notions of role, society, group) and then enabling their management (construction, inspection, adaptation) at both development and execution time. This synergy makes it possible to model and enact coordination activities taking into account the organisation context where they take place, characterised by some structure . in terms of roles, groups, or societies – and organisation rules, such as access control policies. Agents participate to social activities always by virtue of their position (roles) inside the organisation, which define what kind of coordination artifacts they can access and use, and what kind of actions they are allowed (or forbidden) to do on them.

As an example, introduced in [32], the Agent Coordination Context (ACC) abstraction is an infrastructural notion suitable for the integration of organisation issues in a coordination context, especially in the case of artifact-based coordination infrastructure. The ACC notion is meant to model and enact agent position inside an organisational context acting as its environment, so as to define and constrain the agent actions on resources, in this case coordination artifacts [37]. Therefore, it is possible to conceive a MAS infrastructure which fruitfully adopts ACC to model and rule agent presence inside the organisation, and, more specifically, agent participation to social activities; this participation includes accessing and using the coordination artifacts as part of organisation resources.

3

Agent Oriented Infrastructures

3.1 JADE

JADE (Java Agent DEvelopment Framework) [4, 5] is a software framework fully implemented in Java language, and provides an agent-oriented infrastructure. It is based on a middleware compliant with the FIPA specifications and offers a set of graphical tools that supports the debugging and deployment phases.

3.1.1 Static Model

Agents are the main entity in **JADE**, since it is an agent-oriented framework to develop Multi Agent Systems. In part (B) of Figure 3.1 is depicted a fragment of the general version of the meta-model of JADE, extracted from [42]. The class *AgentJ* represents the agent: each agent has an associated *Agent State*, a *Scheduler*, and a *Message Queue*. An agent can be in different states, that represent the possible states listed by FIPA specifications: ACTIVE, DELETED, IDLE, INITIATED, SUSPENDED, TRANSIT, WAITING. Agents can move from one state to another according to the admitted transitions: the possible behaviours can be represented by a finite state machine. The scheduler is implemented by the base Agent class, it is hidden to the programmer, and carries out a round-robin non-preemptive scheduling policy among all behaviours available in the ready queue. Every behaviour is executed until the behaviour releases the control. The message queue is a sort of mailbox associated to every agent: it stores all the messages (written using the Agent Communication Language) sent by other agents and that have to be read by the agent. In the class diagram representing the meta-model, there the class Message Queue is composed by a set of *ACL Messages*.

Behaviour is used to model a generic task, and is specialized into *CompositeBehaviour* and *SimpleBehaviour*. *SimpleBehaviour* models simple tasks,

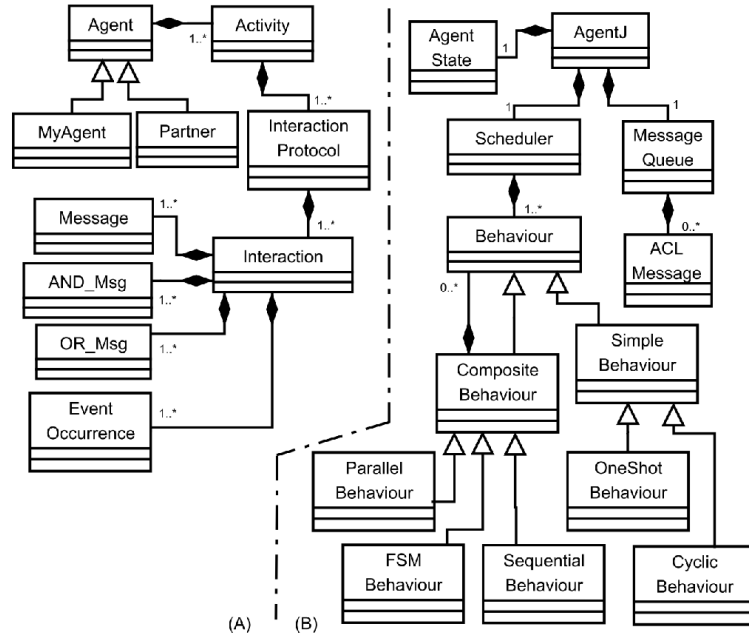


Figure 3.1: JADE Meta-model

namely tasks that are not decomposed into sub-tasks. It is in turn specialized into *OneShotBehaviour* and *CyclicBehaviour*, respectively used to represent tasks to be executed only once and cyclic tasks that are restarted after finishing their execution cycle. *CompositeBehaviour* models complex tasks, that are made up by composing a number of other tasks: in order to represent the necessity of composition there is an aggregation between *CompositeBehaviour* and *Behaviour*, with cardinality 0..*. There are three specializations of the composite behaviours: *FSMBehaviour*, *SequentialBehaviour*, and *ParallelBehaviour*. *FSMBehaviour* is used when the complex task is composed by tasks corresponding to the states of a Finite State Machine; *SequentialBehaviour* is a classical sequential composition of sub-tasks; *ParallelBehaviour* allows the definition of concurrency, where tasks are executed in virtual parallelism.

Figure 3.2 shows a detail of the JADE meta-model focused on the Agent, the key concept of the architecture. Every agent has one or more associated AID (Agent IDentifier) that can be used to reference an agent, and whose role is describing agents. There is also an association towards the class *Codec*, which represents the tool used to codify and decodify ACL messages and is internally called by the JADE framework. A set of agents share an *Agent-Toolkit*, namely the place where they live and execute: a typical example of agent toolkit is the JADE container. Agents also have an associated

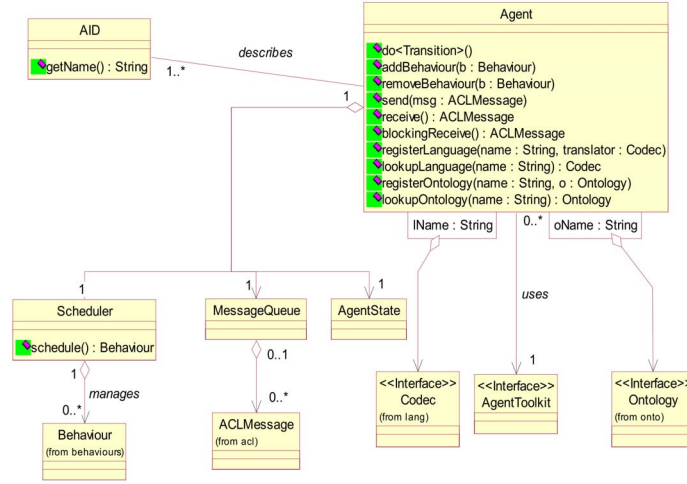


Figure 3.2: **JADE** Meta-model: focus on agent

Ontology, needed to give semantics to the content of the ACL messages agents exchange.

JADE is provided with a content reference model (Figure 3.3), which is needed to perform the proper semantic checks on a given content expression, by identifying all possible elements in the domain of discourse.

The classification is derived from the ACL language specifications. A *Predicate* is an expression that says something about the status of the world and can be either true or false. A *Term* is an expression identifying both abstract and concrete entities, that exist in the world and that the agents talk and reason about. The class *Term* is abstract, and is refined into *IRE*, *Concept*, *Primitive*, *Aggregate*, and *Variable*. An *IRE* (Identifying Referential Expression) is an expression that identifies the entities for which a given predicate is true. A *Concept* is an expression that indicate entities with a complex structure that can be defined in terms of slots. A *Primitive* represents atomic entities such as strings and integers. An *Aggregate* is an expression indicating entities that are a group of other entities. A *Variable* is an expression used to indicate a generic element not known a-priori. As well as the class *Term* is used to represent entities, the class *ContentElement* is introduced to express the content of an ACL message, and the class *Predicate* derives from it. *ContentElementList* is used to indicate a list of content elements. The class *AgentAction* derives from both *ContentElement* and *Concept*. JADE ontologies have to deal with predicates, concepts, and agent actions.

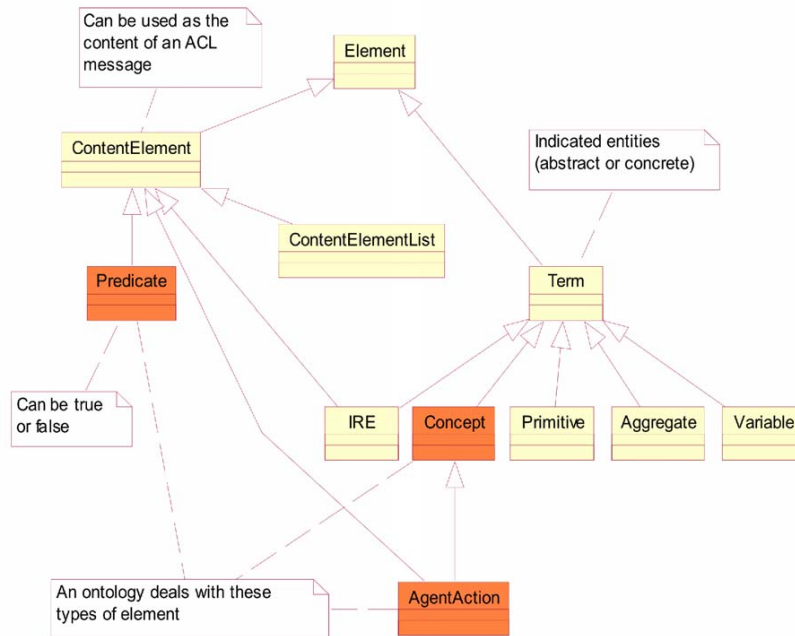


Figure 3.3: **JADE** Meta-model: focus on content

3.1.2 Dynamic Model

From the functional point of view, **JADE** provides the basic services necessary to distributed peer-to-peer applications in the fixed and mobile environment. **JADE** allows each agent to dynamically discover other agents and to communicate with them according to the peer-to-peer paradigm. From the application point of view, each agent is identified by a unique name and provides a set of services: it can register and modify its services and/ or search for agents providing given services, it can control its life cycle and, in particular, communicate with all other peers.

The first service / agent is the *Agent Management System* (AMS) whose responsibility is to manage the life cycle of all agents in the platform, and to provide a white pages service associating logical agent identifier to status and address information [46] as illustrated in Figure 3.4. The agent A asks to AMS the address of a particular agent B. The AMS executes an internal search and produces a result for A. Now A is able to send a FIPA message to B through the infrastructure layer.

In Figure 3.5 is reported an example of the interaction among agent and AMS: AMS must authorise each operation that agent makes inside the in-

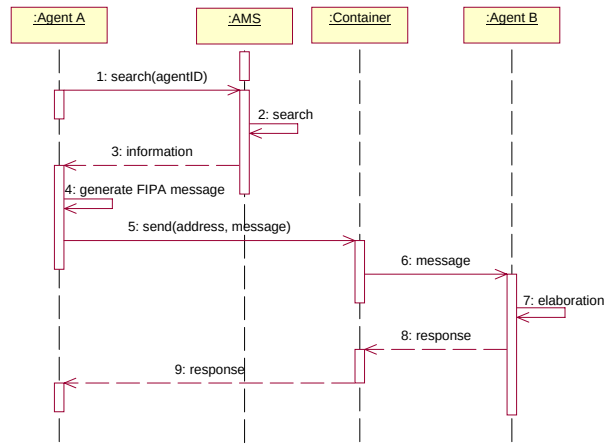


Figure 3.4: Message passing

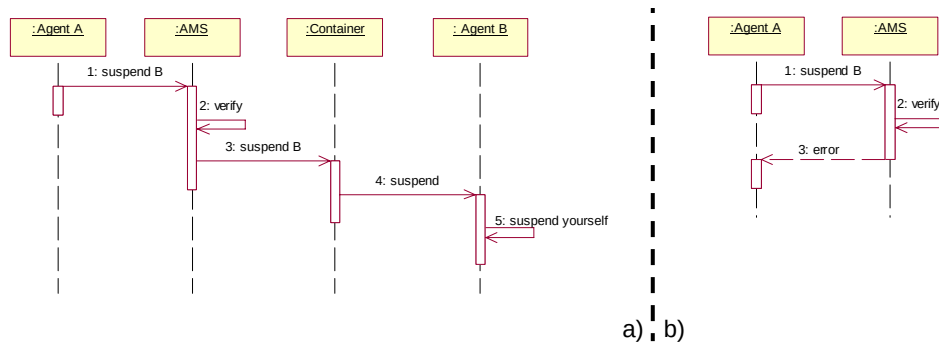


Figure 3.5: The request of a suspension of an agent: a) the authorised case, b) the non-authorised case

frastructure. In the specific case of the Figure 3.5 an agent A asks to AMS the suspension of another agent B. If the AMS authorises this suspension the message is propagate through the infrastructure layer (indicated as *Container* in the figure) to agent B (part a) of the figure); otherwise A is informed that the suspension of B is not possible.

The second service / agent is the *Directory Facilitator* (DF), that works as a yellow pages service with basic matchmaking capabilities. A special instance of this agent – named the *Default DF* – must be available in a FIPA compliant agent platform. An example of this is reported in Figure 3.6 where the agent A contacts the Default DF searching an agent that provides a specific service. The Default DF executes an internal search and produces the information for A. So A is now able to send its request to the agent B.

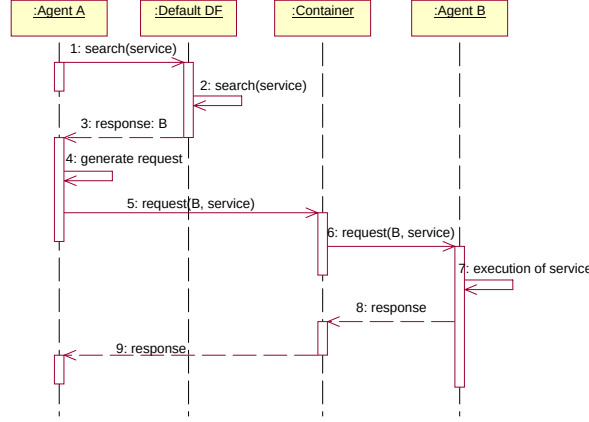


Figure 3.6: Discovery of a service

3.2 TuCSoN

TuCSoN (Tuple Centres Spread over Networks) [40, 3] is an infrastructure providing services for the communication and coordination of distributed / concurrent independent agents.

3.2.1 Static Model

The **TuCSoN**'s meta-model is depicted in Figure 3.7. In detail, **TuCSoN** supports agent communication and coordination via *tuple centres* coordination media [15]: these are shared & reactive information spaces, distributed over the infrastructure *nodes*. In turn, this inducts a *topology* over the *network*. Agents access tuple centres associatively, by writing (**out**), reading (**rd**, **rdp**), and consuming (**in**, **inp**) *tuples* – i.e., ordered collections of heterogeneous information chunks – via the above coordination primitives.

A tuple centre is a tuple space enhanced with the notion of behaviour specification. More precisely, a tuple centre is a coordination abstraction perceived by the interacting entities as a standard tuple space [16], but whose behaviour in response to events can be defined so as to embed the coordination laws. So, defining a new behaviour for a tuple centre basically amounts at specifying state transitions in terms of *reactions* to *events* [40]. In particular, reactions are specified in **TuCSoN** via the **ReSpecT** (Reaction Specification Language) language [33]: a reaction is defined as a set of non-blocking operations [40], and has a success/failure transactional semantics: a successful reaction may atomically produce effects on the tuple centre state, a failed reaction yields no result at all. Typically, a tuple centre contains a

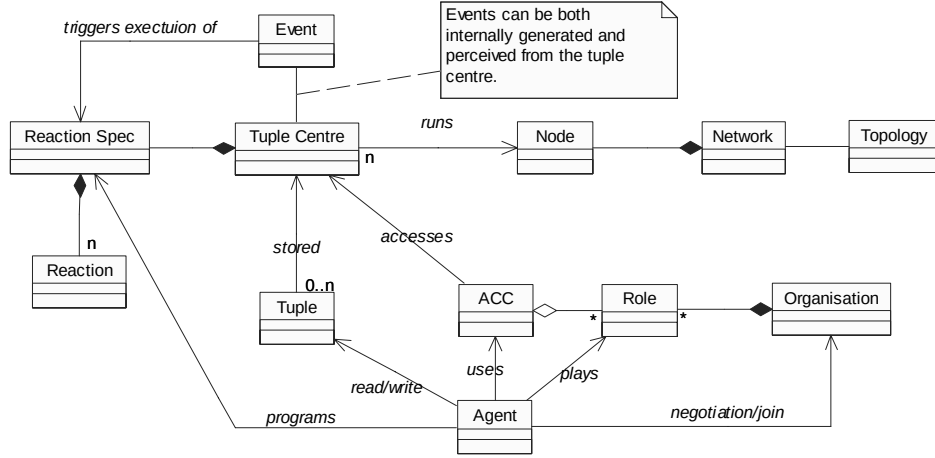


Figure 3.7: TuCSon Meta-model

set of reactions (*reaction spec* in Figure 3.7), each tied to a specific event: the same event could trigger multiple, different reactions. Tuple centres are connected to each other through *link* operations, having the same form and a similar semantics as TuCSon coordination primitives, but invoked by successful reactions rather than by agents [33].

The Agent Coordination Context (*ACC*), introduced in [32] as the conceptual place where to set the boundary between the agent and the environment, encapsulates the interface enabling agent actions and perceptions inside the environment. More precisely, an *ACC* (*i*) works as a model for the agent environment, by describing the environment where an agent can interact, and (*ii*) enables and rules the interactions between the agent and the environment, by defining the space of the admissible agent interactions.

The *ACC* abstraction can be fruitfully exploited to model the presence or position of an agent within an *organisation*, in terms of its admissible actions with respect to organisation resources and its admissible communications toward the other agents of the organisation. The *ACC* abstraction makes it possible to explicitly model the presence of an agent in an organisational context, where specific structures and rules are defined: for the purpose, the *ACC* can embed and reify the relationships that link an individual agent with respect to his position in the structure. In particular, in a role-based organisational context, the *ACC* makes it possible to represent the *role(s)* that an agent is playing inside an organisation, and then his responsibilities, permissions and the interactive behaviour (protocols, conversations). The *ACC* is then particularly suitable in those organisational frameworks where

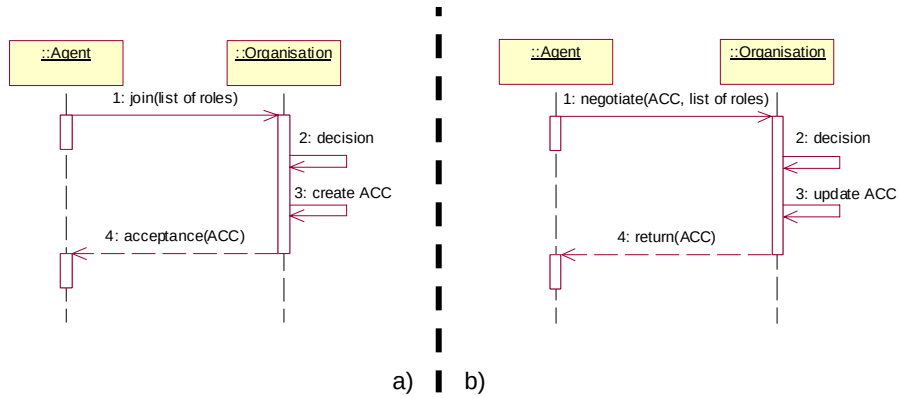


Figure 3.8: Agent joins organisation (a) and agent re-negotiate ACC (b)

the role characterisation is provided solely in terms of the expected or allowed observable behaviour of the agents, not relying on agent internal (e.g. cognitive) aspects [43].

3.2.2 Dynamic Model

The ACC dynamics is characterised by two basic steps: *negotiation* and *use*. In fact, an ACC is meant to be first negotiated by the agents with the MAS infrastructure in order to start a working session inside an *organisation* (Figure 3.8 part a)). To this end, the agent specifies which *roles* to activate: if the agent request is compatible with the (current) organisation rules, then a new ACC is created and configured according to the characteristics of the specified roles, and is released to the agent for active playing inside the organisation (Figure 3.8 part a)). However, an ACC could be re-negotiated by agent during its life cycle (Figure 3.8 part a)). In this case the agent contacts the organisation asking the activation of new different roles. The organisation evaluates the request and if this last is compatible with the organisation rules then the ACC is updated and released again to the agent.

The agent can then use the ACC to interact with other agents in the organisation, and with the organisation environment, by performing the actions and activating the perceptions made possible by the ACC. In particular each action an agent performs must be authorised by ACC (Figures 3.10 and 3.9): if the action is not allowed according to the ACC policy, an exception is generated, preventing action execution(Figure 3.9); otherwise, the action is executed.

Typically an agent action is the execution request of a coordination operation. The coordination operations are the tuple centre coordination prim-

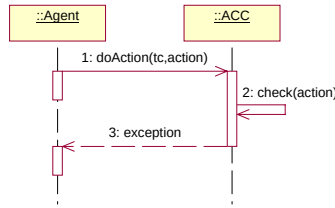


Figure 3.9: Execution of action not authorised by ACC

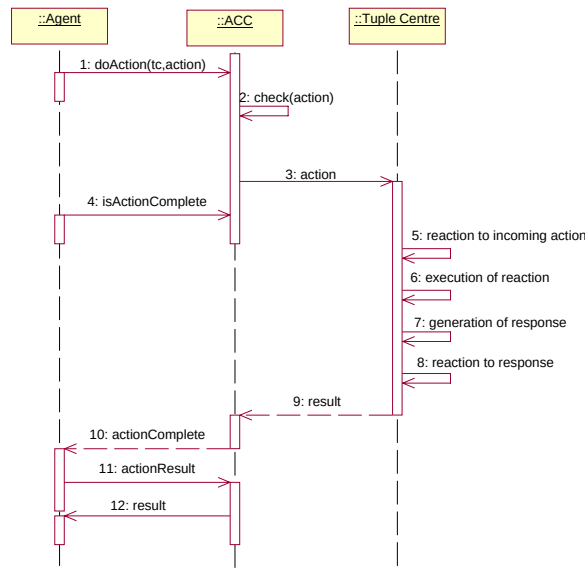


Figure 3.10: Execution of action authorised by ACC

itives, for inserting, reading and retrieving tuples from tuple centre (Figure 3.10). The execution of the incoming action triggers one or more reactions that are executed inside the tuple centre. The execution of reactions generates a partial response in the tuple centre, and this last could trigger other reactions that are executed and generate the final result inside the tuple centre (Figure 3.10 points 5-8). The result is sent to the ACC, and the agent can retrieve it from the ACC.

A particular case of the tuple centre coordination primitives are the primitives for inspecting / changing the behaviour specification of a tuple centre (Figure 3.11). When a primitive of this kind is executed in the tuple centre no reactions are triggered and the tuple centre sends immediately the generated result to the ACC.

A tuple centre is also able to perceive events generated in the network (for

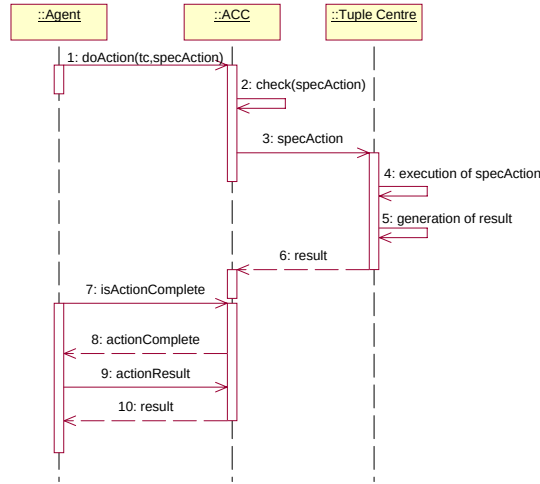


Figure 3.11: Programming of a Tuple Centre

example events generated by other tuple centres or agents). When an event is perceived by a tuple centre, it could react executing the triggered reactions (Figure 3.12 part a)), and eventually it could generate and propagate an event in the network; otherwise no reaction could be triggered by the event (Figure 3.12 part b)).

3.3 MARS

MARS (Mobile Agent Reactive Spaces) implements a portable and programmable Linda-like coordination architecture for Java-based mobile agents [7, 8]. extension.

3.3.1 Static Model

Figure 3.13 depicts the meta-model of **MARS**. Globally, the MARS architecture is made up of a multiplicity of independent *tuple spaces*, each one associated to a *node* and accessed by the *agents* locally executing in that node. Therefore, the only requirement to integrate **MARS** with a mobile agent system is to make the *agent server* - in charge of accepting and executing incoming agents on a node - supply agents with a reference to the local MARS tuple space. Agents on a node can then access the local tuple space via a well-defined set of Linda-like primitives, while each MARS tuple space can react to these accesses with differentiated behaviors, programmed in a *meta-level tuple space* to meet specific needs.

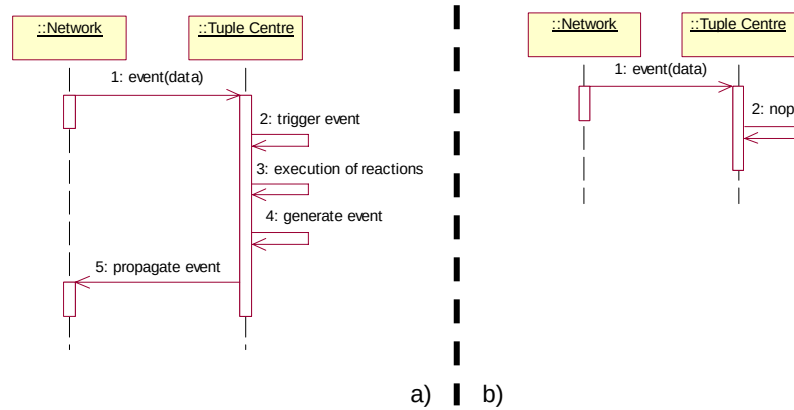


Figure 3.12: Managing of event

Linda [15] adopts three basic primitives: *in*, *rd*, *out*. In particular, *out*(*t*) is used to put a tuple *t* in the tuple space, *in*(*t*) retrieves a tuple *t* from the tuple space and *rd*(*t*) retrieves a copy of *t* from the tuple space (i.e., *t* is still there). The primitives *rd* and *in* are blocking primitives and will suspend execution until the desired tuple has been found, while *out* is nonblocking primitive.

MARS *reactions* are entities, whose implementation code represents the code of the reaction, and are associated to specific access events. Therefore, the association of reactions to access events can be dealt with meta-level 4-ples stored in a meta-level tuple space. A reaction has access to the base-level tuple space and can perform any kind of operation on it (however, operations on the base-level tuple space performed within a reaction do not issue any reaction, to avoid endless recursion). This can be used to exploit the base-level tuple space as a repository of reaction state information, in addition to the state that the reaction object can store as part of its state. As a consequence, the behavior of the reaction can depend both on the actual content of the tuple space and on past access events. Also, when the reaction executes, it has the availability of either the result of the matching mechanism issued by the associated operation, or the template with which an input operation was invoked. Then, the reaction can influence the effects of operations and, for example, can return to the invoking agent different tuples than the ones resulting from the normal pattern-matching mechanisms.

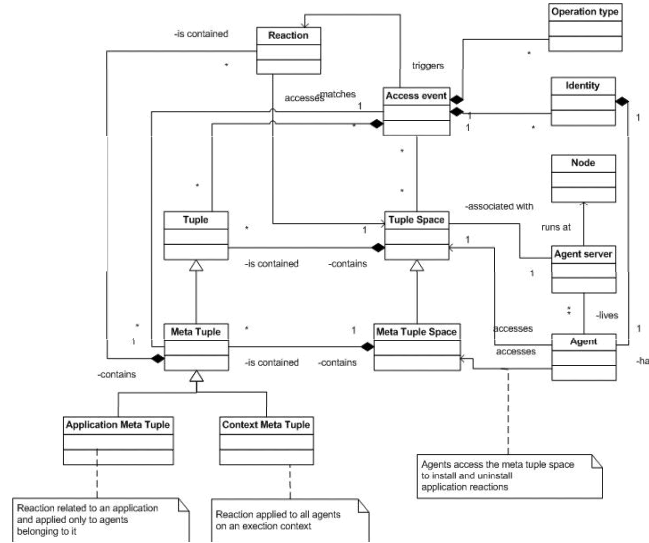


Figure 3.13: MARS Meta-model

3.3.2 Dynamic Model

The **MARS** architecture 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. Globally, the **MARS** architecture is made up of a multiplicity of independent tuple spaces, each one associated to a node and accessed by the agents locally executing in that node. Therefore, the only requirement to integrate **MARS** with a mobile agent system is to make the agent server - in charge of accepting and executing incoming agents on a node - supply agents with a reference to the local **MARS** tuple space. Agents on a node can then access the local tuple space via a well-defined set of Linda-like primitives, while each **MARS** tuple space can react to these accesses with differentiated behaviors, programmed in a meta-level tuple space to meet specific needs.

MARS reactions are associated to specific access events on the basis of the three components that characterize an access event: *tuple item* (T), *operation type* (Op), and *agent identity* (I). Therefore, the association of reactions to access events can be dealt with meta-level 4-ples stored in a “meta-level” tuple space. A meta-level 4-ple has the form of (*Rct*, *T*, *Op*, *I*): it means that the reaction method of the object *Rct* will be triggered when an agent with identity *I* invokes the operation *Op* on a tuple matching

T. Putting and extracting tuples from the meta-level tuple space (either via a graphical administration interface provided with **MARS** or by means of authorized agents) means installing and uninstalling, respectively, reactions associated to events at the base-level tuple space. A meta-level 4-ple can have some non-defined values, in which case it associates the specified reaction to all the access events that match it. For example, the 4-ple (ReactionObj, null, read, null) in the meta-level tuple space associates the reaction of the ReactionObj instance to all read operations, whatever the tuple type and content and the agent identity. The meta-level tuple space follows associative mechanisms similar to the ones of the “base-level” tuple space. A meta-level pattern-matching mechanism is activated for any access to the base-level tuple space, to detect the presence of reactions to be executed (i.e., of meta-level tuples matching the access event) and trigger their execution. A reaction has access to the base-level tuple space and can perform any kind of operation on it (however, operations on the base-level tuple space performed within a reaction do not issue any reaction, to avoid endless recursion). This can be used to exploit the base-level tuple space as a repository of reaction state information, in addition to the state that the reaction object can store as part of its state. As a consequence, the behavior of the reaction can depend both on the actual content of the tuple space and on past access events. Also, when the reaction executes, it has the availability of either the result of the matching mechanism issued by the associated operation, or the template with which an input operation was invoked. Then, the reaction can influence the effects of operations and, for example, can return to the invoking agent different tuples than the ones resulting from the normal pattern-matching mechanisms (Figure 3.14). So, the sequence of performed steps when an agent accesses a **MARS** space is the following:

1. The agent accesses the base tuple space, supplying a template tuple;
2. The base tuple space selects all the matching tuples, and bid the meta-level tuple space;
3. The meta-level tuple space searches for meta tuples associated to the access event;
4. If found, the corresponding reaction is executed, which can (i) access the base tuple space, (ii) modify the tuples, (iii) create new tuples;
5. The meta-level tuple space returns the final tuples to the base tuple space, which returns them to the accessing agent.

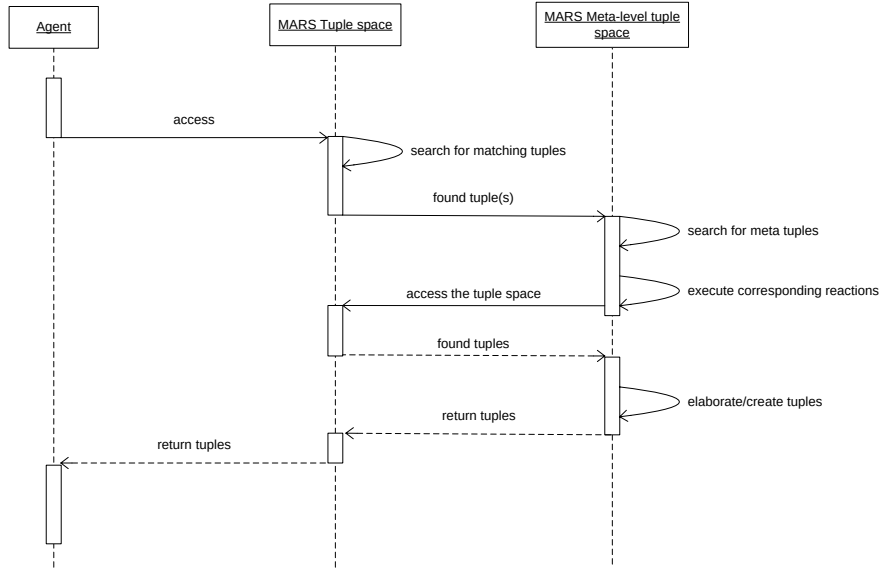


Figure 3.14: MARS

3.4 TOTA

TOTA (Tuples On The Air) [21, 22] is an infrastructure for multi-agent coordination, in distributed computing scenarios.

3.4.1 Static Model

A meta-model of the infrastructure is presented in Figure 3.15. TOTA assumes the presence of a network of possibly mobile *nodes*, each running a *tuple space* [15]: each agent is supported by a local infrastructure and has only a local (one-hop) perception of its environment. Nodes are connected only by short-range *network* links, each holding references to a (limited) set of *neighbour* nodes: so, the *topology* of the network, as determined by the neighbourhood relations, may be highly dynamic.

In TOTA, tuples are not associated to a specific node (or to a specific data space) of the network: rather, they are “injected” in the network by an agent from some node, then autonomously propagate hop-by-hop, diffuse, and evolve according to specified propagation patterns. Thus, TOTA tuples form a sort of spatially-distributed data structure, that can be used to acquire contextual information about the environment and to support the mechanisms required for stigmergic interaction [41]. More precisely, TOTA distributed tuples $T=(C,P,M)$ are characterised by a *content* C , a *propagation rule* P , and a *maintenance rule* M : the content C is an ordered set of

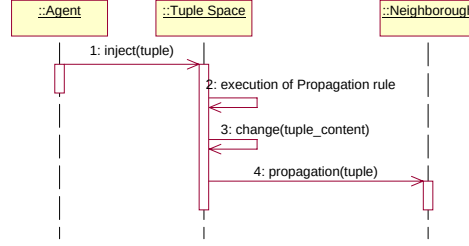


Figure 3.16: Agent injects a tuple

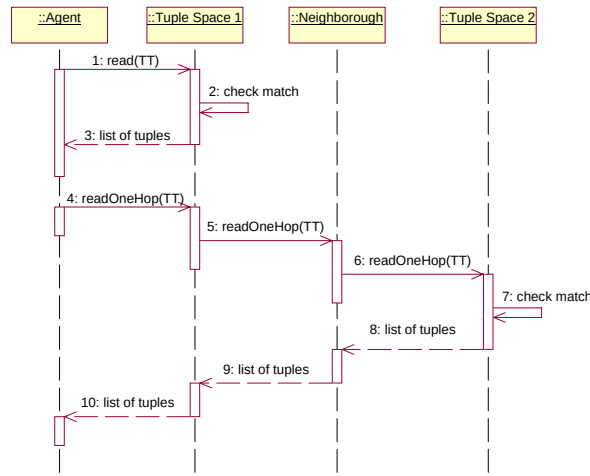


Figure 3.17: Agent reads tuples

Hop action, indeed, returns a collection of the tuples present in the tuple spaces of the node's one-hop neighborhood and matching the template tuple (Figure 3.17 points 4-10).

As mentioned above, the maintenance rule *M* determines how a tuple's distributed structure should react to events occurring in the environment. These types of event can be simple time alarms or they can be events associated to changes in the network structure. To this end, TOTA supports tuples propagation actively and adaptively: by constantly monitoring local events, the network local topology and the income of new tuples, the infrastructure automatically re-shape tuples and their distributed structure whenever appropriate w.r.t. the maintenance rule (Figure 3.18). In particular the tuple space first executes all the maintenance rules associated to the stored tuples, changes the tuples' contents according to maintenance rules and if necessary it propagates the tuples to the other tuple spaces.

In addition, *subscribe* and *unsubscribe* operations are defined in order to

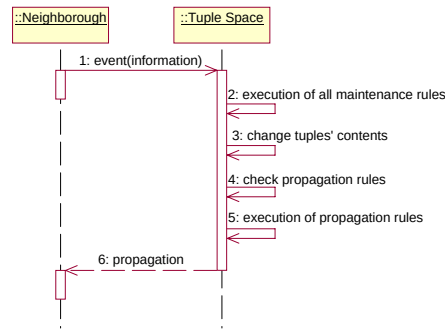


Figure 3.18: Tuple space reacts to events

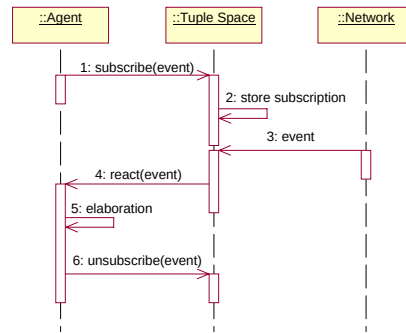


Figure 3.19: Agent subscribes and unsubscribes event

agent can handle events (Figure 3.19). These operations rely on the fact that any event occurring in TOTA (including: arrivals of new tuples, connections and disconnections of peers, system-level events) can be represented as a tuple. Thus: the subscribe operation associates the execution of a reaction method in the agent in response to the occurrence of events matching the template tuple passed. Specifically, when a matching event happens, the tuple space sends the matching event to the agent. The unsubscribe operation removes matching subscriptions.

3.5 RoleX

3.5.1 Static Model

The BRAIN (Behavioural Roles for Agent INteractions) framework [1] proposes an approach where the interactions among agents are based on the concept of role.

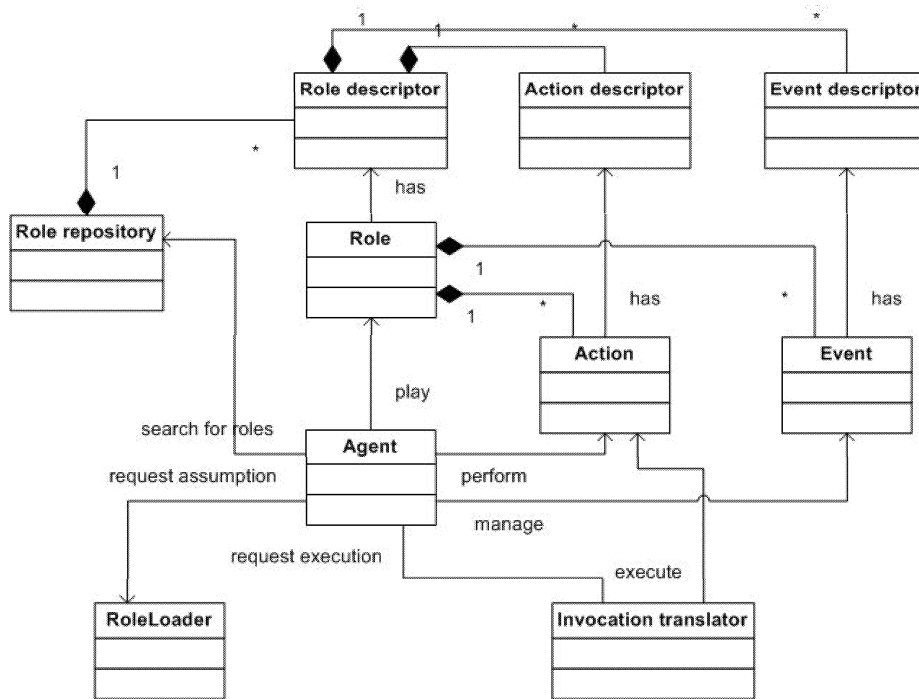


Figure 3.20: **RoleX** Meta-model

In the context of BRAIN, Cabri et al. have implemented an interaction infrastructure [7], called **RoleX** (Role eXtension), whose meta-model is reported in Figure 3.20. **RoleX** can easily be associated to Java agent platforms by adapting the few platform-dependent classes.

In **RoleX**, a role is defined as a set of actions that an agent playing such role can perform to achieve its task, and a set of events that an agent is expected to manage in order to “behave” as requested by the role it plays. Interactions among agents are then represented by couples action-event, which are dealt with by the underlying interaction system, which can enforce local policies and rules.

To grant a high level of abstraction in assuming, playing and discarding roles, **RoleX** exploits *descriptors* for roles, actions and events. A descriptor is an object that describes a role, an action or an event, for example with some keywords, an aim, a version, a creation date and any further needed piece of information. Action descriptors are exploited to associate specific methods to operations. Event descriptors tell the kind and the context of the occurred event, but not how to manage it. Role descriptors describe what such role does but not how (with which operations) it is done. A role descriptor includes also the descriptors of the corresponding actions and

events.

Using descriptors, the agent programmer does not need to know which is the physical class that implements a role, but only the descriptor of the role to be searched for. The descriptors are useful also for hiding to the agent the physical location of the role implementation, allowing agent programmers to disregard about the work of role programmers, and viceversa, because the role behavior is described in a separate way. Descriptors are stored in an appropriate repository which can be queried by agents.

The fact that an agent assumes a role means that the infrastructure dynamically adds each role implementation member (both methods and fields) to agent members, in order to add the set of capabilities of the role, thus modifying and extending the agent class bytecode.

To implement the dynamic role extension, our system is based on a special class loader, called “role loader”, that can change agent behavior and external appearance. The idea is simple: an agent that wants to assume a new role, after querying the role descriptor database, asks the role loader to reload itself with the new role. If everything is right, the role loader sends the agent an event to indicate that the agent has been reloaded. After the reload event the agent can resume its execution. To release a role, the process is the same, but this time the agent is reloaded without that role.

The programmers do not know anything about the role implementation but know by the descriptors, which actions can be used, and which events can occur. In the following, we focus on the action use, because the management of the events is similar and simpler. The use of descriptors means that the programmer cannot write code that invokes methods corresponding to role actions in the usual way, because a compile-time error will occur. Therefore, there must be an invocation translator that do introspection on the extended agent to dynamically find which method must be call in response to an invocation on an action description. When the agent invokes a role action, it specifies to the *invocation translator* a descriptor of the action that wants to perform, the translator searches for a method that corresponds to the description and then invokes it.

3.5.2 Dynamic Model

The management of roles in **RoleX** is highly dynamic, since an agent can assume at runtime roles even unknown at compile time, thanks to a special class loader called **RoleLoader**. Such a high dynamism is also granted by the exploitation of *descriptors*, which uncouple the physical implementation of a role (i.e., the Java class) from its features.

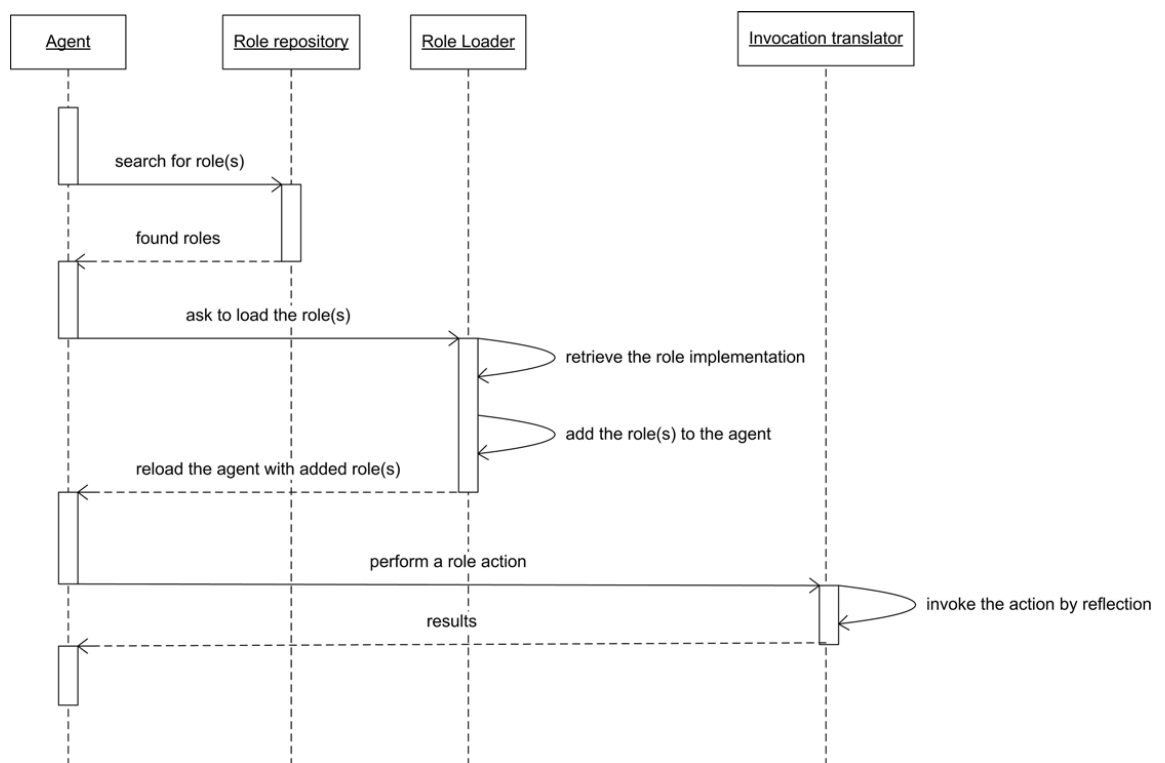


Figure 3.21: Agent assumes a role and performs actions

In **RoleX**, the fact that an agent assumes a role means that the infrastructure, via the already-mentioned *RoleLoader*, dynamically adds each role implementation member (both methods and fields) to agent members, in order to add the set of capabilities of the role, thus modifying and extending the agent class bytecode. The sequence of activity performed in **RoleX** when an agent assumes roles and perform actions is reported in Figure 3.21.

When an agent wants to assume one or more roles to carry out its tasks, it queries the role repository in order to find the best role(s) for its needs. Then, the agent asks the *RoleLoader* to reload itself with the new role (or, better, with the role members added to the agent's one). This step can be repeated if the found roles are more than one. If everything is right, the *RoleLoader* sends the agent an event to indicate that the agent has been reloaded. After the reload event the agent can resume its execution.

The programmers do not know anything about the role implementation but know by the descriptors, which actions can be used, and which events can occur. In the following, we focus on the action use, because the management of the events is similar and simpler. The use of descriptors means that the programmer cannot write code that invokes methods corresponding to role actions in the usual way, because a compile-time error will occur. Therefore, there must be an *invocation translator* that does introspection on the extended agent to dynamically find which method must be call in response to an invocation on an action description. When the agent invokes a role action, it specifies to the invocation translator a descriptor of the action that wants to perform, the translator searches for a method that corresponds to the description and then invokes it using the reflection.

To release a role, the process is similar to the addition, but this time the agent is reloaded without that role.

3.6 CARTAGO

CARTAGO [9, 45] is a framework for developing artifact-based working environments for multi-agent systems. The framework is based on the notion of artifact, as basic abstraction to model and engineer objects, resources and tools designed to be used and manipulated by agents at runtime to support their working activities, in particular the cooperative ones. **CARTAGO** makes it possible for MAS engineers to design and develop suitable kind of artifacts, and to extend existing agent platforms with the possibility to create artifact-based working environments, programming agents to exploit them.

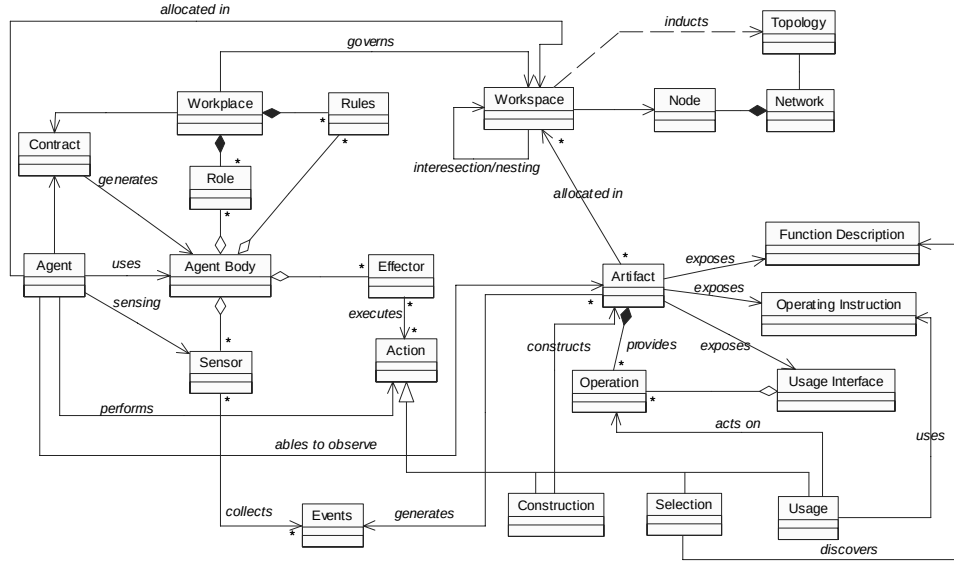


Figure 3.22: **CARTAGO** Meta-model

3.6.1 Static Model

The abstract architecture of **CARTAGO** (Common Artifact for Agents Open environment) is composed of three main elements (see Figure 3.22): (i) *agent bodies* – as the entities that make it possible to situate agents inside the working environment; (ii) *artifacts* – as the basic building blocks to structure the working environment; and (iii) *workspaces* – as the logical containers of artifacts, aimed at defining the topology of the working environment.

Agent bodies. The agent body contains *effectors* to perform *actions* upon the working environment, and a dynamic set of *sensors* to collect *events* from the working environment. Agents interact with their working environment by “piloting” their bodies: they execute actions to *construct*, *select* and *use* artifacts, and perceive the *observable events* generated by artifacts.

Artifacts. Artifacts are the basic bricks managed by **CARTAGO**: agents use artifacts by triggering the execution of *operations* listed in the artifact *usage interface*. The execution of an operation typically causes the update of the internal state of an artifact, and the generation of one or more observable events: these are then collected by the agent sensors and perceived by means of explicit sensing actions. In order to support a rational exploitation of artifacts by intelligent agents, each artifact is equipped with a *function description*, i.e. an explicit description of the functionalities it provides, and *operating instructions*, i.e. an explicit description of how to use the artifact to get its function.

Workspaces. Artifacts are logically located in workspaces, which define the *topology* of the working environment. A workspace is an open set of artifacts and agents: artifacts can be dynamically added to or removed from workspaces by agents, agents can dynamically enter (join) or exit workspaces. Workspaces make it possible to structure agents and artifacts organisation & interaction: in particular, workspaces can function as scopes for event generation and perception, as well as for artifact access and use. Articulated topologies can be created via workspace intersection and nesting: in particular, intersection is supported by allowing the same artifacts and agents to belong to different workspaces.

In addition, **CARTAGO** also introduces the concept of *workplace* as an organisational layer on top of workspaces. More precisely, a workplace is the set of *roles* and *organisational rules* being in force in a workspace: there, *contracts* define the norms and policies that rule agent access to artifacts and allow the generation of agent bodies. So, for instance, an agent may or may not be granted permission to use some artifacts or to execute some specific operations on selected artifacts depending on the role(s) that the agent is playing inside the workplace [44].

3.6.2 Dynamic Model

CARTAGO provides two *default artifacts* in each workspace in order to support agents in the creation and discovery of artifacts. In particular the *Factory Artifact* creates new artifacts and the *Registry Artifact* provides a white page services.

In order to create an artifact, an agent uses the Factory Artifact (Figure 3.23 part a)) providing to it several information about the new artifact: a logic name, the template that identifies the type of the artifact to be created, the initial configuration parameters and optionally the workspace where the artifact should be created. The action can fail if the template is unknown or the artifact instantiation is not completed due to some kind of problem. In order to search an artifact, an agent uses the Registry Artifact (Figure 3.23 part b)) providing the logic name of the artifact. Then the Registry Artifact sends the artifact identifier to the agent.

In order to execute an operation over artifact, an agent must first joins the workspace where the artifact is allocated, then a workspace produces the relative agent body (Figure 3.25 points 1-3). By means of agent body the agent is able to use an artifact specifying the artifact identifier, and the operation name (Figure 3.25 points 5-7).

The artifact executes the requested operation, eventually changes its state

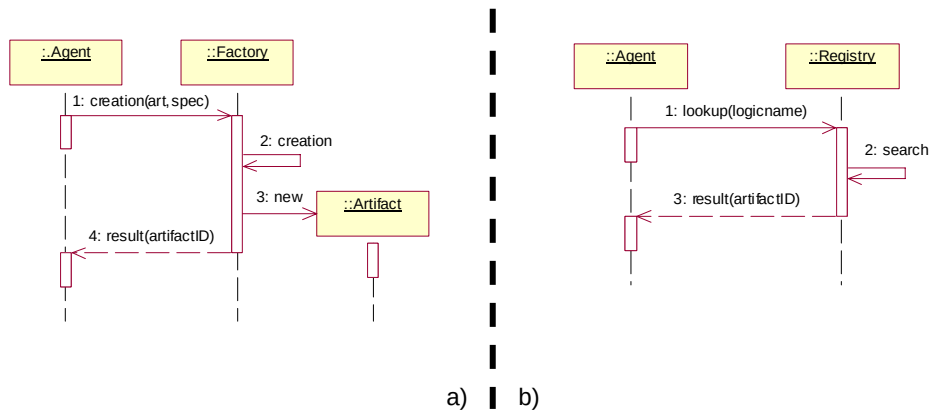


Figure 3.23: Agent creates a new artifact (a) and agent search an artifact

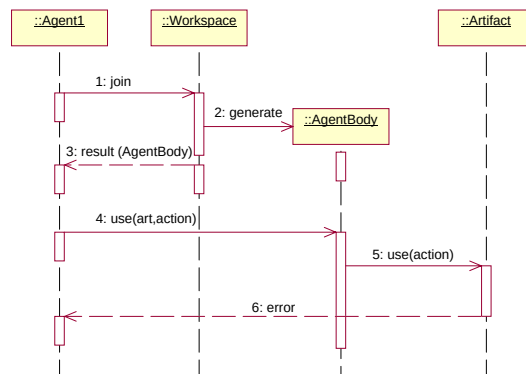


Figure 3.24: Problems in the use of artifact

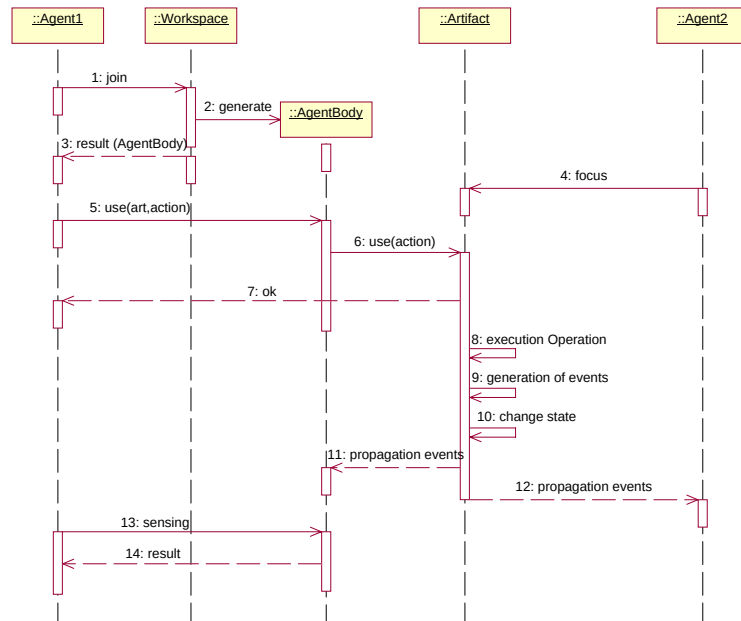


Figure 3.25: Agent uses an artifact

and the generates one or more events propagated to the agent body (Figure 3.25 points 5-11). The action can fail either because the specified artifact is not available, or because the operation cannot be executed since it is not part of artifact usage interface (as illustrated in Figure 3.24). Action success means that the execution of the specified operation has been successfully triggered. After that the agent is able to access to the operation results by means of a sensing operation (Figure 3.25 points 13-14).

However an agent could observe all the event produced by a specific artifact without use it. In this case the agent manifests this intention by means of the *focus* action: each time the artifact generates an event, this event is propagate to the agent (Figure 3.25 points 4,12).

4

Conclusions

MAS infrastructures play a fundamental role in the engineering of complex software systems. Their role is at least as fundamental as the one of methodologies as discussed in Chapter 2. Infrastructure is a fundamental notion for complex systems in general, not only in computer science and engineering, but also in the context of organisational, political, economical and social sciences. In this work, we provided a brief overview over current conceptualisations, models and support infrastructures for MAS.

The importance of infrastructure is even more evident when considering agents and MAS as main paradigm, whose typical scenarios involves complex systems composed by a multitude of distributed active entities organised according to some model, immersed in specific software / hardware environments, executing some kind of individual and collective tasks that require their fruitful interaction and coordination. Complexity of such scenarios can be framed at different levels of abstraction: from concurrency and distribution (control, space, time,...), to system / environment openness in terms of dynamisms, heterogeneity, unpredictability, and so on. As in the case of human societies, infrastructures in MAS are meant to play a fundamental role in governing such a complexity, factorising critical issues of systems in terms of services at different levels, from sustainability to security, from communication to coordination and organisation, and so on.

Currently, most of the available state of the art MAS infrastructures has been conceived and designed in academic contexts, and can be still considered in their infancy, in particular with respect to mainstream middlewares. Among the others, such an immaturity affects three different aspects: *(i)* abstractions and services provided *(ii)* available tools and technologies *(iii)* integration with agent-oriented methodologies. Then, we consider the research on MAS infrastructure models and supporting tools, on the integration with agent-based methodologies, and on architectures and technologies used for MAS development essential for advancing the state of the art on software engineering, in particular for what concerns complex software systems.

Bibliography

- [1] Agent and Pervasive Computing Group. BRAIN home page. <http://www.agentgroup.unimo.it/MOON/BRAIN>.
- [2] aliCE Research Group. SODA home page. <http://soda.alice.unibo.it>.
- [3] aliCE Research Group. TuCSon home page. <http://tucson.sourceforge.net>.
- [4] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Developing multi-agent systems with a fipa-compliant agent framework. *Softw., Pract. Exper.*, 31(2):103–128, 2001.
- [5] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE: a fipa2000 compliant agent development environment. In *Agents*, pages 216–217, 2001.
- [6] Carole Bernon, Massimo Cossentino, Marie Pierre Gleizes, Paola Turci, and Franco Zambonelli. A study of some multi-agent meta-models. In James Odell, Paolo Giorgini, and Jörg P. Müller, editors, *Agent-Oriented Software Engineering V*, volume 3382 of *LNCS*, pages 62–77. Springer, 2004. 5th International Workshop, AOSE 2004. Revised Selected Papers.
- [7] Giacomo Cabri, Luca Ferrari, and Letizia Leonardi. Enabling mobile agents to dynamically assume roles. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 56–60, New York, NY, USA, 2003. ACM Press.
- [8] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. MARS: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000.
- [9] CARTAGO. Home page. <http://www.alice.unibo.it:16080/projects/cartago/>.
- [10] John Charles. Middleware moves to the forefront. *Computer*, 32(5):17–19, 1999.
- [11] Enrico Denti, Andrea Omicini, and Alessandro Ricci. Coordination tools for MAS development and deployment. *Applied Artificial Intelligence*, 16(9–10):721–752, October–December 2002. Special Issue: Engineering Agent Systems – Best of “From Agent Theory to Agent Implementation (AT2AI-3)”.
- [12] Dov Dori. *Object-Process Methodology: A Holistic System Paradigm*. Springer, 2002.
- [13] Foundation for Intelligent Physical Agents. FIPA home page. <http://www.fipa.org>.

- [14] Les Gasser. Mas infrastructure: Definitions, needs and prospects. In Thomas Wagner and Omer F. Rana, editors, *Agents Workshop on Infrastructure for Multi-Agent Systems*, volume 1887 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2001. International Workshop on Infrastructure for Multi-Agent Systems, Barcelona, Spain, June 3-7, 2000, Revised Papers.
- [15] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
- [16] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.
- [17] Joseph Andrew Giampapa and Katia Sycara. Team-oriented agent coordination in the RETSINA multi-agent system. Technical Report CMU-RI-TR-02-34, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, December 200. Presented at AAMAS 2002 Workshop on Teamwork and Coalition Formation.
- [18] IBM. WebSphere home page. www-3.ibm.com/software/ts/mqseries.
- [19] Stanley M. Sutton Jr. Middleware selection. In Wolfgang Emmerich and Stefan Tai, editors, *Engineering Distributed Objects, EDO 2000*, volume 1999 of *Lecture Notes in Computer Science*, pages 2–7. Springer, 2000. Second International Workshop, EDO 2000, Davis, CA, USA, November 2-3, 2000, Revised Papers.
- [20] Anna Liu and Ian Gorton. Accelerating cots middleware acquisition: The i-mate process. *IEEE Software*, 20(2):72–79, 2003.
- [21] Marco Mamei and Franco Zambonelli. Programming stigmergic coordination with the TOTA middleware. In Frank Dignum, Virginia Dignum, Sven Koenig, Sarit Kraus, Munindar P. Singh, and Michael Wooldridge, editors, *Proceedings of AAMAS 2005*, pages 415–422. ACM Press, 25–29 July 2005.
- [22] Marco Mamei and Franco Zambonelli. Programming modular robots with the tota middleware. In Hideyuki Nakashima, Michael P. Wellman, Gerhard Weiss, and Peter Stone, editors, *5th Inter. Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006)*, pages 485–487. ACM, 2006.
- [23] Microsoft. .Net home page. www.microsoft.com/net.
- [24] Ambra Molesini, Enrico Denti, and Andrea Omicini. MAS meta-models on test: UML vs. OPM in the SODA case study. In Michal Pěchouček, Paolo Petta, and László Zolt Varga, editors, *Multi-Agent Systems and Applications IV*, volume 3690 of *LNAI*, pages 163–172. Springer, 2005. 4th International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'05), Budapest, Hungary, 15–17 September 2005, Proceedings.

- [25] Ambra Molesini, Enrico Denti, and Andrea Omicini. From aose methodologies to mas infrastructures: The soda case study. 2007. Accepted to the 8th International Workshop “Engineering Societies in the Agents World” - October 22-24, 2007 in Athens - Greece.
- [26] Ambra Molesini, Andrea Omicini, Enrico Denti, and Alessandro Ricci. **SODA**: A roadmap to artefacts. In Oğuz Dikenelli, Marie-Pierre Gleizes, and Alessandro Ricci, editors, *Engineering Societies in the Agents World VI*, volume 3963 of *LNAI*, pages 49–62. Springer, 2006. 6th International Workshop (ESAW 2005), Kuşadası, Aydın, Turkey, 26–28 October 2005. Revised, Selected & Invited Papers.
- [27] Ambra Molesini, Andrea Omicini, Alessandro Ricci, and Enrico Denti. Zooming multi-agent systems. In Jörg P. Müller and Franco Zambonelli, editors, *Agent-Oriented Software Engineering VI*, volume 3950 of *LNCS*, pages 81–93. Springer, 2006. 6th International Workshop (AOSE 2005), Utrecht, The Netherlands, 25–26 July 2005. Revised and Invited Papers.
- [28] Pablo Noriega and Carles Sierra. Electronic institutions: Future trends and challenges. In Matthias Klusch, Sascha Ossowski, and Onn Shehory, editors, *Cooperative Information Agents VI*, volume 2446 of *Lecture Notes in Computer Science*, pages 14–17. Springer, 2002. 6th International Workshop, CIA 2002, Madrid, Spain, September 18-20, 2002, Proceedings.
- [29] Object Management Group. CORBA home page. <http://www.corba.org>.
- [30] Object Management Group. UML home page. <http://www.uml.org>.
- [31] Andrea Omicini. **SODA**: Societies and infrastructures in the analysis and design of agent-based systems. In Paolo Ciancarini and Michael J. Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *LNCS*, pages 185–193. Springer-Verlag, 2001. 1st International Workshop (AOSE 2000), Limerick, Ireland, 10 June 2000. Revised Papers.
- [32] Andrea Omicini. Towards a notion of agent coordination context. In Dan C. Marinescu and Craig Lee, editors, *Process Coordination and Ubiquitous Computing*, chapter 12, pages 187–200. CRC Press, Boca Raton, FL, USA, October 2002.
- [33] Andrea Omicini. Formal **ReSpecT** in the A&A perspective. *Electronic Notes in Theoretical Computer Sciences*, 175(2):97–117, June 2007. 5th Inter. Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA’06), CONCUR’06, Bonn, Germany, 31 August 2006. Post-proceedings.

- [34] Andrea Omicini and Sascha Ossowski. Objective versus subjective coordination in the engineering of agent systems. In Matthias Klusch, Sonia Bergamaschi, Peter Edwards, and Paolo Petta, editors, *Intelligent Information Agents: An AgentLink Perspective*, volume 2586 of *LNAI: State-of-the-Art Survey*, pages 179–202. Springer-Verlag, March 2003.
- [35] Andrea Omicini, Sascha Ossowski, and Alessandro Ricci. Coordination infrastructures in the engineering of multiagent systems. In Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli, editors, *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*, volume 11 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, chapter 14, pages 273–296. Kluwer Academic Publishers, June 2004.
- [36] Andrea Omicini and Alessandro Ricci. Reasoning about organisation: Shaping the infrastructure. *AI*IA Notizie*, XVI(2):7–16, June 2003.
- [37] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Formal specification and enactment of security policies through Agent Coordination Contexts. *Electronic Notes in Theoretical Computer Science*, 85(3):17–36, August 2003. 1st International Workshop “Security Issues in Coordination Models, Languages and Systems” (SecCo 2003), Eindhoven, The Netherlands, 28–29 June 2003. Proceedings.
- [38] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Coordination artifacts as first-class abstractions for MAS engineering: State of the research. In Alessandro F. Garcia, Ricardo Choren, Carlos Lucena, Paolo Giorgini, Tom Holvoet, and Alexander Romanovsky, editors, *Software Engineering for Multi-Agent Systems IV: Research Issues and Practical Applications*, volume 3914 of *LNAI*, pages 71–90. Springer, April 2006. Invited Paper.
- [39] Andrea Omicini and Giovanni Rimassa. Towards seamless agent middleware. In *IEEE 13th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2004)*, pages 417–422, 2nd International Workshop “Theory and Practice of Open Computational Systems” (TAPOCS 2004), Modena, Italy, 14–16 June 2004. IEEE CS. Proceedings.
- [40] Andrea Omicini and Franco Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, September 1999.
- [41] H. Van Dyke Parunak. “go to the ant”: Engineering principles from natural agent systems. *Annals of Operation Research*, 75:69–101, 1997.
- [42] Loris Penserini, Anna Perini, Angelo Susi, and John Mylopoulos. From stakeholder intentions to software agent implementations. In Eric Dubois and Klaus

- Pohl, editors, *Advanced Information Systems Engineering*, volume 4001 of *Lecture Notes in Computer Science*, pages 465–479. Springer, 2006. 18th International Conference, CAiSE 2006, Luxembourg, Luxembourg, June 5-9, 2006, Proceedings.
- [43] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. Agent coordination context: From theory to practice. In Robert Trappl, editor, *Cybernetics and Systems 2004*, volume 2, pages 618–623, Vienna, Austria, 13–16 April 2004. Austrian Society for Cybernetic Studies. 17th European Meeting on Cybernetics and Systems Research (EMCSR 2004), 4th International Symposium “From Agent Theory to Theory Implementation (AT2AI-4)”. Proceedings.
 - [44] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. **CARTAgO**: An infrastructure for engineering computational environments in MAS. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *3rd Inter. Workshop “Environments for Multi-Agent Systems” (E4MAS 2006)*, pages 102–119, 8 May 2006.
 - [45] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. **CARTAgO**: A framework for prototyping artifact-based environments in MAS. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *Environments for Multi-Agent Systems*, volume 4389 of *LNAI*, pages 67–86. Springer, February 2007. 3rd Inter. Workshop (E4MAS 2006). Selected Revised and Invited Paper.
 - [46] Giovanni Rimassa. *Runtime support for distributed multi-agent systems*. PhD thesis, University of Parma, 2003.
 - [47] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.
 - [48] Kjeld Schmidt and Carla Simone. Coordination mechanisms: Towards a conceptual foundation of CSCW systems design. *Computer Supported Cooperative Work*, 5(2/3):155–200, 1996.
 - [49] Sun Microsystems. J2EE home page. <http://java.sun.com/j2ee>.
 - [50] Katia P. Sycara, Massimo Paolucci, Martin Van Velsen, and Joseph A. Giampapa. The RETSINA MAS infrastructure. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):29–48, 2003.
 - [51] M. Tambe, D. Pynadath, N. Chauvat, A. Das, and G. Kaminka. Adaptive agent integration architectures for heterogeneous team members. In *ICMAS*, pages 301–308. IEEE Computer Society, 2000. 4th International Conference on Multi-Agent Systems (ICMAS 2000), 10-12 July 2000, Boston, MA, USA.
 - [52] Milind Tambe and Weixiong Zhang. Towards flexible teamwork in persistent teams: Extended report. *Autonomous Agents and Multi-Agent Systems*, 3(2):159–183, 2000.

- [53] TEAMCORE Group. TEAMCORE home page. <http://teamcore.usc.edu/>.
- [54] Jos van Hillegersberg, Kuldeep Kumar, and Richard J. Welke. Using meta-modeling to analyze the fit of object-oriented methods to languages. In *31st Hawaii Inter. Conference on System Sciences (HICSS 1998)*, volume 5: Modeling Technologies and Intelligent Systems, pages 323–332, Kohala Coast, HI, USA, January 6-9 1998. IEEE Computer Society.
- [55] Mirko Viroli and Andrea Omicini. Coordination as a service: Ontological and formal foundation. *Electronic Notes in Theoretical Computer Science*, 68(3):457–482, March 2003. 1st International Workshop “Foundations of Coordination Languages and Software Architecture” (FOCLASA 2002), Brno, Czech Republic, 24 August 2002. Proceedings.