Engineering the Environment of Multiagent Systems

Mirko Viroli, Alessandro Ricci ({mirko.viroli,a.ricci}@unibo.it)
DEIS, Alma Mater Studiorum – Università di Bologna, Cesena, Italy

Franco Zambonelli (franco.zambonelli@unimore.it)
Dipartimento di Scienze e Metodi dell'Ingegneria, Università di Modena e Reggio Emilia, Italy

Tom Holvoet, Kurt Schelfthout
({tom.holvoet,kurt.schelfthout}@cs.kuleuven.be)
AgentWise, DistriNet, Katholieke Universiteit Leuven, Belgium

Abstract. Whereas the notion of environment is first-class in agent-based systems by definition, its role and its potential in conceiving and developing multiagent applications has only recently been recognised.

In this paper, we argue for the need of tackling the environment as a key concern in engineering multiagent systems. Considering the environment as a separate abstraction is a recognition of its importance, and creates the field of engineering MAS environments. We bring together existing work on middleware and infrastructure for MAS, and we present the concept of "environment abstraction" as the basic building block for one approach to engineer the environment. Environment abstractions represent functional components, which can be composed to constitute the environment. We clarify how existing middleware and infrastructure relates to environment abstractions, and we outline research challenges both in this approach as well as towards alternative paths for engineering the environment.

Keywords: multiagent systems, agent-oriented software engineering

1. Introduction

Software engineering always calls for exploiting proper *abstractions* for modelling and developing the various components of a software system: abstractions such as functions, objects, components, data items and their relations have e.g. dominated mainstream software engineering practices so far. A cornerstone of agent-oriented software engineering (AOSE) is the idea that the emerging characteristics of modern, complex software systems call for more powerful abstractions, accounting for the fact that (**?**; **?**): *(i)* software is fruitfully modelled as a composition of autonomous software entities, whose behaviour can be understood and designed in terms of "achieving a goal"; *(ii)* such entities do not live and execute in isolation, but are always situated for execution in a computational and/or physical environment. Despite the environment notion in any basic definition of agent, mainstream theories, platforms, and methodologies promoted by the agent community almost entirely neglect the environment as a key constituent of agent-based systems.

According to standard software engineering practice, engineering such a quite complex entity calls for a careful treatment (**?**; **?**) which ultimately requires the notion of environment to be elected as first-class in engineering multi-agent systems, impacting nearly all stages of development — design, implementation, and run-time.

Recognizing the environment as a valuable abstraction is one thing, modeling and developing the software that shapes the environment is another. In fact, the environment can be a complex entity of the system. Its responsibilities are numerous, including ..., and need to be seamlessly integrated. Engineering such complex entities involves the several phases of development, including software architecture, detailed design, implementation, and execution.

One, although generic, approach to engineer the environments is discussed in this paper. This approach recognizes the concept of an "environment abstraction" as a basic building block for modeling the functional decomposition of the environment. Such a model of the environment can be considered the software architecture of the environment, as it describes the early design decisions of this software entity and describes the functionalities and relationships between the environment abstractions.

Yet it is useless to propose a new generic abstraction without considering the large body of research that has been conducted on middleware and infrastructure for MAS. Therefore, we provided an overview of this research, and clarify and illustrate how existing middleware and infrastructure can support the environment abstractions.

We open the perspective on engineering the environment by recognizing that different approaches for modeling and decomposing this complex entity are worth investigating. In particular, advances in research on software engineering, including research on software architectures and research on aspect-oriented development, aim to better support the development of large and complex distributed systems in general. We briefly discuss how this research could proof its value for engineering the environment of MAS, and allude to research challenges in these areas.

The paper is structured as follows. In Section 2, we analyse the role of the environment in engineering multiagent systems, focussing on its relevance and impact as both a deployment context for agents — a set of legacy physical and computational components the agent should be designed to exploit — and as a design dimension — when crucial system parts are explicitly built and encapsulated in the environment.

We then identify two main areas pertaining (in different ways) the issue of engineering the environment of multiagent systems. The first one, described in Section 3, discusses the environment abstraction and middleware and infrastructure for MAS for supporting modeling and developing an environment model. Independently of the details of the specific multiagent application of interest, there is often the need to factorise an abstract model

of the environment into a separate *middleware*. This middleware can take different forms, but can in general be considered as a software layer supporting the environment model, providing the designer with an infrastructure of services and functionalities to be exploited when developing a specific multiagent application. We describe several such middlewares, each of which tackles a diverse nature and role of the environment — knowledge sharing, designed coordination, emergent coordination, and organization.

The second area, described in Section 4, is about the impact of the environment in the development of a multiagent system application. Most existing AOSE (agent-oriented software engineering) methodologies currently lack the notion of environment, and time is premature for devising a unified and conceptually general integration. On the other hand, research in software engineering could also proof its usefulness in developing complex applications as MAS. Both points of view are discussed.

We conclude in Section 5 outlining main directions for future work in the context of engineering the environment for multiagent systems.

2. The Role of the MAS Environment in Software Engineering

2.1. Complexity, Agents and Environment

In order to manage complexity, software engineers leverage three fundamental tools: decomposition, abstraction and organisation (or hierarchy) (**?**). Briefly, a complex system can be understood, designed and then implemented — in one word, engineered — by *abstracting* away from its low-level details, and considering it as *decomposed* in loosely coupled constituents, reflecting a specific *organization* where each such constituent plays a specific role and has given interrelationships with others. The agent-based approach to software engineering has been claimed to be a good paradigm to leverage these tools (**?**): a complex system is decomposed in agent organizations, each made of a set of interacting agents with given interrelationships. The agent abstraction plays a key role in this context, promoting the viewpoint of a system in terms of autonomous entities in charge of proactively achieving a goal (**?**; **?**), namely, a sub-goal of the whole application.

Therefore, the multiagent system approach makes applications and their engineering different from traditional scenarios. In the object-oriented perspective, for instance, the "everything is an object" perspective dominates, and a software systems is globally conceived as a composite functional architecture of interacting and passive objects, all subject to a sort of centralized flow of control (**?**; **?**). On the other hand, multiagent systems identify the agents in charge of accomplishing specific tasks in autonomy — thus relying on abstractions of a higher level.

From their definition, agents do not live and execute in isolation, but are always situated for execution in an *environment*. This is either a "physical"

environment which agents perceive and act upon through proper sensors and actuators, and/or a "computational" one composed of other software constituents, namely, digital resources such as knowledge repositories, information sources, service providers, and so on (**?**). However, whereas classically agents are seen as the true source of complexity in a multiagent system — even in (**?**) the notion of agent is given primary importance over that of environment — a deeper look reveals that successful multiagent applications take their advantage of complex environments.

Although most of the investigations on environments consider primarily weak forms of agency (**?**; **?**), there is potentially a strong interplay between environments and cognitive agents, i.e. strong agency. Supporting evidence can be found in theories that focus on human societies, such as Activity Theory (**?**) and Distributed Cognition (**?**). Also research in AI and DAI, in particular the work of Agre (**?**), Rosenschein (**?**), Kirsh (**?**), Dourish (**?**) — along with recent works in cognitive sciences, such as the work of Clark and Chalmers (**?**) —, clearly remark the fundamental impact that the environment can have on supporting agent reasoning, changing the way agent and agent societies solve problems and execute tasks.

This consideration drives us to the need of more carefully applying *decomposition*, *abstraction*, and *organization*, electing the environment to a first-class constituent in the engineering of a multiagent system, along with agents and at the same level. Consequently, we argue that multiagent systems engineering should feature as a key part the process of *(i)* decomposing a multiagent system into agents and environment, *(ii)* devising the proper abstract model of the environment, and *(iii)* focussing on the interrelationships between agents and environment, as well as between the inner constituents of the environment structure.

This idea contrasts the typical argument that the presence of different classes of entities — agents and environment components — is an additional source of complexity, and should be tackled via a process of "agentification" where everything that is not an agent — namely, the environment constituents — is wrapped inside additional agents, acting as sorts of autonomous managers. The systematic adoption of agentification tends instead to produce a conceptual abstraction mismatch without actually diminishing complexity: it applies the agent abstraction to model real world concepts that do not fit the main characteristics of agents. It is preferable to leverage the distinct nature of agents and environment towards finding the solution to a software problem.

The environment can be seen as composed by a set of entities which are not autonomous and goal-oriented like agents. They are instead "passive", in the sense that their function in the multiagent system is exploited by agents perceiving them and acting upon them: in other words, *the environment constituents are used by the agents and never the opposite.* This calls for a completely different characterisation: features like autonomy, proactivity,

social attitude and goal-driven behaviour simply do not apply to the environment, which thus requires different abstract models, and potentially a different engineering approach.

However, the distinction between the agent level and the environment level may not always be clear cut. When interpreting an application as a multiagent system, one can reactive or semi-active entities that cannot be sharply considered as purely autonomous or purely passive, and they could therefore be seen either as agents or as parts of the environment. When engineering a system, some tasks can be identified that could be either realised in an autonomous or proactive way by some agent, or can be automatised and realised for agent exploitation by a suitably-engineered passive component of the multiagent system environment. In general, understanding what entities are to be modeled as passive and part of the environment and what should be agents is something that may require an accurate analysis that ultimately depends on the problem's characteristics.

2.2. The Environment as a Deployment Context

In the current scenario of today's applications, software systems are rarely developed to be deployed as stand-alone, isolated systems. In most of the cases, software systems — and multiagent systems specifically — are conceived for execution in an already existing system of components (**?**) which we call the *deployment context*, and which could be either computational or physical.

Trivially, even the simplest system conceived for a single workstation or a PC, has to execute on a specific operating system, rely on existing shared libraries, and properly exploit the existing structure of the file system. More generally, modern distributed applications are built to live and interact in an existing world of data and services, and have to get advantage of them. The sharper example is that of multiagent systems for Web-based applications, in which agents are deployed in the Web and are in charge of mining Web data and exploiting available services in order to achieve specific goals — e.g., organizing a trip for the user by discovering appropriate tourist information and by booking flights and hotels as needed (**?**). Other scenarios exist or have been envisioned which share similar characteristics, such as the Grid (**?**), P2P Networks (**?**; **?**), computational markets (**?**). The perspective is always that of deploying agents or multiagent systems that have to start executing by properly accessing and acting on existing digital resources and services.

On the other hand, embedded, pervasive, and mobile computing technologies are making today's systems more and more strictly inter-twined with the physical world as well. Embedded (multiagent) systems for the control of manufacturing processes are being intensively deployed, to take care of sensing physical characteristics and of affecting the production and

transformation of physical goods (**?**). Pervasive computing systems, such as sensor networks (**?**) and RFID tags (**?**) allows to acquire in a digital form (and put to service of software) a variety of information about the surrounding environment. Mobile computing technologies, enabling us to stay connected 24/7 from wherever, require context-awareness and context-dependency, to have computer-supported activities properly adapted to the context and situation from which we are performing them (**?**). In addition to that, an increasing deal of attention is being paid to mobile robotics systems, as sorts of physical agents in charge of performing context-aware physical activities in an environment (**?**). The impact of the above technologies on multiagent systems engineering is that agents have to carry on their activities by continuously processing (and being continuously affected by) what is happening in the physical world.

Independently of the specific case one considers, it is rather clear that the design and development of a multiagent system cannot abstract from the characteristics of its deployment context, whether it is made of computational or physical resources. Thus, the necessity of exploiting the environment necessarily comes into play simply because such environment exists as a legacy component, which does not relief designers from a careful modelling effort.

2.3. The Environment as a Design Dimension

The deployment context surely forms a crucial environment for multiagent systems, with a key role in applications. This scenario has been in fact subject of research investigation since the beginning of AI and DAI, in particular for what concern situated multiagent systems (**?**; **?**).

However, it is not always the case that the environment is something already existing, which the engineer is simply subject to as it is and has to leverage at his/her better to achieve the overall system goals. Instead, as emphasised in many recent research projects and industrial systems (see the "Application" paper in this volume), many parts of the environment can and should be explicitly designed and implemented during the multiagent system development.

A notable example is the case of stigmergy, inspired by biological systems such as ant colonies (**?**; **?**). In order to facilitate the interaction of mobile agents in distributed systems, stigmergic infrastructures are built which allow agents to deposit and then perceive *pheromones* — represented in terms of chunks of information. These are diffused in the distributed system by the infrastructure and eventually evaporate, allowing agent interaction according to spatial and temporal locality criteria. Examples of approaches adhering to this scenario include TOTA (**?**), Parunak's work in (**?**), and the PROSA architecture (**?**). Other examples of explicitly designed environments for multiagent systems include infrastructures for general

purpose coordination (?), e-institutions architectures (?), middlewares for organization (?) — and others that will be described in next section.

Moreover, even when the multiagent system is immersed in a significant deployment context of digital resources, it is generally fruitful to wrap them in an explicitly designed environment. A main advantage of this approach is to raise the abstraction level of such external resources: agent-to-environment interaction can be understood and designed at the knowledge or social level (?; ?) instead of a low level call to an external and existing service infrastructure. The work on infrastructure integration in (?) or the work on resource artifacts (?) are example researches along this direction.

3. Supporting the Environment

Since the environment has to be treated as *first-class* in the engineering of multiagent systems, this impacts all the stages of the application development process, from design-time to implementation-time and run-time. Since different aspects of environment support arise in each stage, we discuss each in turn.

Design — One of the main tasks of the design stage is to identify a general software model of the environment. In this model, the environment is typically to be seen as decomposed in several constituents which we call *environment abstractions*.The term "abstraction" is used here basically as a synonym for "component" or "constituent", and emphasizes the need for a proper abstraction over the actual implementation details — the knowledge level is e.g. a proper abstraction level to use (?). This notion of *environment abstraction* can take different forms and incarnations according to the functionalities and responsibilities that are charged upon the environment. In spite of such variability, being first-class means that such abstractions are used in synergy with the agent abstraction, forming the two basic building blocks to engineer multiagent systems. Accordingly, different kinds of models exist to support the design level, providing basic environment abstractions and architectures to be exploited along with the agent models and platforms chosen for the specific multiagent system.

Implementation — Being first-class at implementation-time means that the abstractions used in the design stage are to be supported in the actual programming of multi-agent systems, by means of suitable libraries, frameworks and languages. So, while there exist libraries, frameworks and languages to program agents, there are also similar support tools to program environment abstractions. For instance, as a language such as 3APL can be used to program goal-oriented agents (?), a language like ReSpecT can be used to specify the coordinating behaviour of *tuple*

Figure 1. MAS layers with environment-based supports

centres — which can be framed as environment abstractions used for coordination (**?**). As a JADE library can be used to develop FIPA compliant agents in Java to be executed on top of JADE platform (**?**), analogously the TOTA library can be used to enable agents developed in Java to exploit the TOTA middleware of tuple spaces for handling computational fields in a network-like logical environment (**?**).

Run-time — Finally, platforms, infrastructures and middlewares are often built to "keep abstractions alive at run-time": they form distributed run-time environments responsible of environment lifecycle and overall management, providing services (API) for accessing, using, and adaptating the supported abstractions. For instance, as the RETSINA MAS infrastructure manages the lifecycle of RETSINA agents, and e.g. basic services for agent direct communication and discovery (**?**), TuCSoN environment infrastructure (**?**) manages the lifecycle of tuple centres, providing services for their access and use by (heterogeneous) agents.

Keeping abstractions alive at run-time is also important for application maintenance and evolution. Run-time observability and controllability are essential properties that software engineers look for when deploying software systems, but in the case of agent-based software engineering, agent autonomy and system openness make the effective achieving of these properties challenging. Environment abstractions can play a fundamental role here: they are the natural loci of observation, testing and control. For example, in the coordination artifacts framework artifacts are meant to explicitly feature a *testing* and *management interface*, with operations to test for the correctness of its behaviour and to dynamically observe and control it.

Figure 1 shows a logical view of typical system layers in a multiagent system, adapting the one in (**?**) to emphasise the relationship between agent and environment abstractions, and their support. The MAS application level is composed by specific application agents and an implementation of the environment abstractions which interact with one another. This is the level at which the engineer of a specific MAS application typically works. Below, there is the execution layer featuring the middlewares and infrastructures supporting agents and environments. Whereas different infrastructures can be used for agents and environment, they have an overlapping area of integration (**?**) that handles "interface" aspects such as management of actions, perceptions and general interaction between agent and environment. Lower layers take into account the deployment context, including legacy software infrastructure, operating system, hardware and the physical world.

In this paper, we focus on the MAS middleware layer, and more specifically describe how several environment middlewares and infrastructures are able to support environment abstractions at the application layer. Methodological aspects concerning how the engineer of a specific MAS application can exploit such abstractions will be considered in Section 4.

3.1. Common Properties of Environment Abstractions

Although different kinds of environment abstractions can be conceived, corresponding to the different kinds of functionalities and responsibilities they are charged with, some common features can be identified:

- *Agent-Environment Interaction* – if communication is the best way to characterise inter-agent direct interaction, this is not the case for the interaction between agents and environment. Rather, this is best framed — as in the basic agent definition, actually — in terms of actions and perceptions: typically, agent actions correspond to the execution of an *operation* provided by the environment, and in response, the environment generates events which can be perceived by the agents. From an higher level point of view, an agent *uses* the services provided by the environment, in order to achieve its goals.
- *Mediation* – All environment abstractions realise a form of *mediation*, either among agents or among agents and (physical, legacy) resources. Such a mediation role is the key for enabling observation, control, and management of the dependencies and the interaction that occur among the agents or among agents and resources. We will show how from an engineering point of view, this can be exploited to design and encapsulate mediation strategies for a wide range of purposes — as will be clear in next subsection —, ranging from concurrency management, up to the management of social dependencies and the enforcement of social norms.
- *Composition and Topology* – Figure 1 shows that the environment abstractions can be linked together. On the one hand, this is useful to reflect the physical or logical topology of the environment — e.g. when the environment itself is distributed. On the other hand, as already mentioned, decomposition is key to handle complexity of environments: different functions of the environment can be encapsulated in different environment abstractions, promoting separation of concerns and a better scaling with complexity.

Figure 2. Overview of some state-of-the-art approaches introducing a notion of environment in MAS

3.2. From Abstractions to Infrastructures

The goal of this section is to consider some of the main scenarios where the environment plays a crucial role in multiagent systems, and describe the main features of the corresponding environment abstractions, examples of specific models adopted, and the related middleware support.

It is worth noting here that the terms "middleware" and "infrastructure" play an almost interchangeable role in the context of supporting the environment of a multiagent system. In distributed systems, middleware is generally defined as the software layer that lies between the operating system and the applications on each site of the system (**?**). Also in multiagent systems middleware is given this meaning (**?**; **?**). On the other hand, the term infrastructure in MAS (**?**; **?**; **?**) more specifically focuses on the idea of supporting services and related functionalities in a distributed software system. In this paper we use both terms: "middleware" is mostly used to stress the structure of the layer supporting the environment, and "infrastructure" for the resulting services as provided to agents and multiagent system engineers.

PROPOSAL: remove this figure + references to it. It is not clarifying, and is not referenced or used any further in the text. As a reference framework, Figure 2 provides a comprehensive (even if not exhaustive) overview of models and corresponding middlewares as found in well known research approaches on environment, with a possible classification according to the functionality provided — this classification should not be intended as a strict classification, some models can be classified in different ways according to the different perspective adopted. A more detailed discussion of the characteristics of some of these approaches is reported in the following. *PROPOSAL: give here an overview + explanation of the catergorization used in the following subsections.*

3.2.1. *Environment-Based Communication and Knowledge Sharing*

Environment abstractions can be used to design and support various kinds of agent interaction besides direct communication, such as *indirect* communication and *implicit* communication, providing some degrees of *uncoupling* that are particularly effective for open and dynamic domains.

An important example is given by *tuple-based* approaches. Generally speaking, in such approaches communication is supported by shared information spaces (tuple spaces), where agents can insert, read and retrieve information chunks (tuples) by means of a basic set of space primitives. Various families of this approach exist, with different models and languages used to encode knowledge in tuples (first-order logic, XML, records, Java

Figure 3. The AGV Application Layers

objects, etc.), different ways to structure the spaces, different extensions to the basic set of primitives (adding for instance event oriented functionalities such as publish/subscribe) — a comprehensive survey can be found in (**?**). As a main property, this form of environment-based communication provides temporal and spatial uncoupling, i.e. the capability of easily supporting the communication between agents that are not simultaneously present and that do not know each other or each other's location. For these reasons, this approach is seen as particularly useful for engineering mobile agent applications. In particular, tuple spaces can be considered as the environment abstractions provided to agents by an infrastructure, with services for creating, destructing, and accessing tuple spaces — see e.g. (**?**).

Besides communication, i.e. knowledge exchange, the environment can be used to support *knowledge sharing*: environment abstractions can be designed as repositories of agent shared knowledge, providing specific functionalities for their exploitation and management, including observation, search, and update. Tuple-based approaches naturally support forms of knowledge sharing.

Instead of being merely passive repositories, such abstractions can be designed and implemented so as to provide specific functionalities which become essential for supporting agent awareness and the coordination of their activities. As an example, we can consider the AGV application described in (**?**), where unmanned vehicles controlled by agents transport various kinds of loads through a warehouse. An environment abstraction called "virtual environment" is used to keep a consistent and updated map of the physical environment (such as the vehicles' location) on the one hand. On the other hand, it encapsulates important functionality to support the coordination of agents. Agents can e.g. put marks in the virtual environment to avoid collisions, and the virtual environment takes on the burden of handling the interaction protocol between various AGVs in the mobile network to synchronize these marks. Figure 3 represents the AGV application, with the various layers in evidence. *ObjectPlaces* (**?**) is the middleware adopted to implement the distributed environment abstraction for the agents operating in a mobile network. By using this middleware, the designer can define interactions in terms of roles, and can define conditions under which such interactions need to be started between the distributed virtual environment instances on the various AGVs. The middleware then activates these interactions given the current situation.

3.2.2. *Environment-Based Designed Coordination*

Multiagent system coordination is perhaps one of the aspects where the environment abstraction has been most studied and adopted in the engi-

Figure 4. The distributed Workflow Management application on top of TuCSoN

neering of applications. For coordination here we mean its most general acceptation coming from organisational sciences, that is the management of dependencies among autonomous actitivies (**?**). The mediation role of environment abstractions makes them the ideal place where to encapsulate functionalities to manage agent dependencies, and effectively support a wide range of agent coordination activities.

Generally speaking, we can divide the environment-based approaches in two different families, which reflect different ways to engineer the systems: *designed coordination* and *emergent coordination*.

In the case of designed coordination, the environment abstractions are designed to encapsulate some kind of coordination functionality or service explicitly defined at design-time (for instance in terms of coordination rules or laws), and that can be suitably exploited by agents at run-time. So, the coordination strategies are explicitly designed by engineers, and are encapsulated and enacted by some kind of environment abstraction. In literature, the notion of *coordination artifact* has been introduced to model and generalise over this kind of approach (**?**). Coordination artifacts can be understood as *tools* that societies of agents can *use* to execute their social activities. On the one side such entities are the basic bricks that can be used to engineer agents' cooperative working environments, on the other side in competitive contexts they can be used to encapsulated and enforce "the rules of the game". Most of the existing environment-based coordination approaches can be described in terms of coordination artifacts design and exploitation.

An example support for this model is given by TuCSoN infrastructure (**?**). The abstractions provided by TuCSoN are called *tuple centres*, which can be framed as kinds of coordination artifacts. Technically, tuple centres are programmable logic based tuple spaces (**?**): they are general purpose logic-based coordination artifacts whose coordinating behaviour can be dynamically specified and customised using the ReSpecT logic language. The TuCSoN infrastructure supports the creation, access and manipulation of tuple centres, distributed among the infrastructure nodes. The inspiring principles of the coordination artifact notion come from socio-psycological theories developed in the context human society, Activity Theory and Distributed Cognition in particular (**?**; **?**).

Designed coordination approaches are useful when it is possible to clearly identify the rules that characterise the social activities — independently of the specific agent behaviours and tasks — which can then encapsulated in suitable environment abstractions separated from the agents participating to the social tasks. An application example over TuCSoN is given by distributed Workflow Management Systems (WfMs) (**?**), in particular in open and dynamic scenarios. In a WfMS, workflow engine components embed

Figure 5. The Intelligent Museum Application on top of TOTA middleware

and enact the workflow rules — defined by designers — for the coordination of the tasks assigned to workflow participants, which are meant to execute such tasks autonomously. This kind of applications can be suitably engineered as a multiagent system where agents play the role of the workflow participants, and workflow engines are designed and built as coordination artifacts, embedding and enacting the workflow rules. Adopting TuCSoN, workflow engines can be implemented on top of tuple centres, programmed so as to enact the coordinating behaviour described by the workflow rules. Figure 4 shows WfMS as a MAS application, with TuCSoN used at the middleware layer. Compared to other agent-based solutions, this approach to WfMS provides a better separation of concerns between individual level and social level, and therefore a better encapsulation.

3.2.3. *Environment-Based Emergent Coordination*

Differently from the designed coordination case, in the case of emergent coordination the environment abstraction does not directly encapsulate the laws defining the coordination activities — which are typically hard or impossible to define — but provide specific mechanisms that promote the emergence of coherent coordinated global behaviour. These approaches typically take inspiration from models and theories coming from physics or biology. Two main kinds of approach are *stigmergy* and *field-based* coordination. In the first case, the environment abstractions take the shape of pheromone environments, supported at runtime by some pheromone infrastructure (**?**), providing services for injecting and perceiving pheromones, and functionalities to program the laws for diffusion, aggregation, and evaporation of such pheromones. In the second case, which can be considered to some extent a particular case of the former as remarked in (**?**), the environment based abstractions represent sorts of fields of forces, influenced by agents actions and influencing agent perceptions.

Typical applications of stigmergy and field-based approaches are used for the engineering of mobile agent applications, in particular in domains requiring the frequent dynamic adaptation of the coordination and movement strategies, following the general self-organisation approach. As an application example we consider the "intelligent museum" application of field-based coordination, which is a system supporting the visitors of a museum in retrieving information about art pieces, to orientate inside the museum, and meet other people in the case of organised groups. This application can be suitably engineered on top of a situated multiagent system using a field-based computational environment, supported by a middleware such as TOTA (**?**). Each agent can inject in the net tuples encapsulating rules for diffusion and evaporation, which can later be perceived by other agents. The

Figure 6. A Market Application on top of AMELI middleware

distribution of such tuples really models a field — like e.g. the gravitation one — which can be navigated to retrieve resources and/or other agents. Figure 5 shows the intelligent museum as a MAS application, on top of TOTA middleware.

3.2.4. *Environment-Based Organisation*

One of the most challenging issues concerning the engineering of agent-based systems is balancing agent autonomy and social order (**?**), in particular in the context of *open* multiagent systems. Open multiagent systems can be regarded as complex systems where (possibly) large and heterogeneous populations of agents interact which exhibit possibly deviating or even fraudulent behaviours. This issue can be tackled by introducing some kind of *regulatory structure* establishing permission and denials for agents, namely, a *normative* system (**?**).

As in the case of coordination, the mediation role of environment abstractions makes them a possible and effective place where to encapsulate and enact norms and — more generally — organisational rules. Taking inspiration from human societies, Electronic Institutions (**?**) implements one such vision, defining the *institution* as first-class abstraction shaping the environment where agents interact, and where the norms are specified and enacted. The relationship with coordination is evident, since norms can be formulated as rules constraining agent interaction, a definition that is often use for coordination (**?**). An example of infrastructure supporting Electronic Institutions is AMELI (**?**). AMELI can be framed as a computational normative environment, general purpose — in the sense that the same infrastructure can be deployed to realise different institutions — and architecturally neutral. An application example is given by markets, where traders (buyers and sellers) meet to trade their good under the supervision of trade manager agents, and where auctions are used as trader protocols. The institution as computational environment can be effectively adopted to specify and enforce the norms that are characteristics of the trade. Figure 6 shows a market as a MAS application, where at the middleware layer is the AMELI infrastructure.

Other examples of environments for normative and organised systems include the abductive logic-based SOCS model (**?**), the work on roles as AGRE (**?**) and Agent Coordination Contexts (**?**), and even the MMASS framework for physical organization (**?**).

4. The Environment in MAS Application Development

The notion of the environment in MAS obviously impacts the development of applications using MAS. We describe two complementary aspects of dealing with the environment in application development. First, we indicate how AOSE methodologies, which are tailored for application development using MAS, can be extended in order to incorporate the environment in the development process. This allows the application developer to explicitly relate functional and non-functional requirements to architectural decomposition. Second, we illustrate the added-value of exploiting advances in software engineerin research (in casu, software architectural approaches) to engineering the environment.

4.1. The Role of the Environment in (Agent-Oriented) Software Engineering

Disciplined software engineering methodologies for agents in general are in their infancy, and extending these techniques to include environments is quite challenging. Most of the AOSE methodologies proposed so far, such as MASE (**?**), TROPOS (**?**) and the earlier version of Gaia (**?**), suggest modelling and building multiagent systems simply as sets of interacting agents, and do not pay any attention at modelling the environment in which agents are situated. As a result, they are not suitable for engineering those applications that can take advantage of the environment abstractions as seen in previous section. Two notable exceptions are SODA (**?**) and GAIA v.2 (**?**).

PROPOSAL: instead of both, choose SODA or GAIA and discuss one of them in more detail, linking much more explicitly to the middlewares discussed in sect. 3. As it is now, neither of them is clear.

Yet, few exceptions exist. One methodology that explicitly cope with the environment is SODA (**?**). Soda takes the environment into account and provides special abstractions and procedures for the design of environment infrastructures. The environment is defined as the space in which agents operate and interact, and is formed by abstract resources — namely, environment abstractions — providing services: the environment model maps such services to infrastructure classes. An infrastructure class is characterised by the services, the access modes, the permissions granted to roles and groups, and the interaction protocols associated to its resources. Infrastructure classes can be further characterised in terms of other features: their cardinality (the number of infrastructure components belonging to that class), their location (with respect to topological abstractions), and their owner (which may be or not the same as the one of the agent system, given the assumption of decentralized control).

Another methodology that explicitly takes the environment into account is Gaia v.2. (hereafter Gaia) (**?**). Modelling the environment in Gaia involves

determining all the entities and resources that the multiagent system can exploit, control or consume when it is working towards the achievement of the organizational goal. The identification of the environment model is part of the analysis phase and is intended to yield an abstract, computational representation of the environment in which the multiagent system will be situated — namely, the deployment context. During the subsequent architectural design phases, the output of the environmental model is integrated in the system's organizational structure that includes the real-world organization (if any) in which the multiagent system is situated. Such an integration leads to a final model for roles, interactions and organizational rules. During the detailed (and final) design phase, the definition of the agent model and services model are derived from the completed role and interaction models. As (**?**) recognises the difficulty in providing universally applicable modelling abstractions and techniques for the environment, a possible general approach is proposed. The environment is described in terms of abstract computational resources, such as variables or tuples, made available to the agents for sensing (e.g. reading their values), for acting (e.g. changing their values) and for consuming (e.g. extracting them from the environment). As such the environmental model is represented as a list of resources, each associated with a symbolic name, characterized by the type of actions that the agent can perform on it and possibly associated with additional textual comments and descriptions.

A key point of both these methodologies, which should be included in any methodology aimed at considering the environment as a first-class notion, is the attempt of finding a general model for the environment abstractions, to be used from design to implementation.

4.2. The Role of Software Engineering in Environment Engineering

The environment constitutes an important part of the overall MAS application. Applications for which MAS are employed are typically quite complex applications: issues including distribution, fault-tolerance, security are to be integrated with the aspects of the environment as a model of the deployment context and with the aspects of the non-trivial mediating and coordinating functionality of the environment as a design dimension. For some applications, the abstractions offered by middleware and infrastructure do not suffice for describing the software architecture for the environment. In that case, the MAS designer can in the design stage be helped by architectural support.

This section looks at the insights of software architecture (**?**) on this non-trivial engineering task. To quote Paul Clemens: "Architecture is design, but not all design is architecture. That is, many design decisions are left unbound by the architecture and are happily left to the discretion and

good judgment of downstream designers and implementers." This line of research aims to systematically deal with various, often conflicting functional and non-functional application requirements. An architecture captures the most essential early design decisions, according to the priorities that the application stakeholders impose.

In this perspective, a multi-agent system model of an application is considered a model of a *solution* for a particular application, encompassing particular stakeholder requirements. Both types of components in such a model - agents and environment - need to be engineered from the architectural analysis phase onward.

A software architecture is defined as "the structure or structures of the system, each of which comprises elements, the externally visible properties of those elements, and the relationships among them". A software architecture typically consists of multiple views (**?**; **?**). Four typical views are the following. The logical view describes the component model, the process view describes the concurrency and synchronization aspects, the physical view describes the mapping of the software onto the hardware and shows the system's distributed aspects, and the development view describes the software's static organization in the development environment.

ROOM FOR EXAMPLE OF VIEWS FOR AGV CASE (avaliable in techrep) ?

An important form of support at the architectural level is reference architectures. A reference architecture defines an abstract architecture that serves as a blueprint to develop software architectures for a family of applications that are characterized by specific functional and quality requirements. Here an explicit, but generic decomposition of the system is given that can be tailored towards specific applications in the reference architecture's chosen application domain. A reference architecture can also be supported further by a middleware or a framework.

Two examples of reference architectures briefly illustrate this. One reference architecture that incorporates a first class environment abstraction, as well as guidelines for designing its internals, is the reference architecture for situated MAS described in (**?**). The reference architecture is applied in a real world application of automatic guided vehicle control. In this application, the concrete software architecture was realized on top of the ObjectPlaces middleware.

A second example of a reference architecture is the MAS reference architecture for manufacturing control, called PROSA (**?**). This reference architecture defines the basic agent types, their functionalities and their relationships. Coordination in PROSA can be achieved through so-called "delegate MAS", i.e. fine-grained agents (called ant agents) which roam the environment to retrieve and distribute information about other agents and resources. The PROSA agents as well as the delegate MAS share the same environment, which is modeled as a directed graph.

If an environment is to be a first class abstraction in the MAS application, it should be incorporated in the reference architecture as well. The FIPA reference architecture for MAS does not take an explicit environment into account, instead e.g. relying on brokers and direct interaction for discovery and coordination.

In conclusion, from the software architectural point of view, an important contribution for environments in MAS would be the definition of appropriate reference architectures for environments, and for MAS with an environment. Although several reference architectures incorporate an abstraction of environment in the MAS architecture, there is room for a more detailed architectural description of the "inner" structure of the environment abstraction. Suitable starting points for such an endeavor can be the middleware and infrastructures described earlier in this paper. Another way to approach this is aspect-oriented software development. Aspect-orientation aims to modularize crosscutting concerns, as well as systematic mechanisms for integrating these concerns. For environment engineering, concerns that typically crosscut the system components (agents and environment) include coordination, distribution, security, fault-tolerance. Research in this area is promising but still in its infancy (**?**; **?**).

5. Conclusions

In this paper we surveyed the main issues that raise when considering the environment as a first-class abstraction in the engineering of multiagent systems. In spite of the infancy of this issue and of general solutions to date, several consolidated frameworks and practices can be identified, as well as open research directions towards a better integration of the agent and environment notion. These include *(i)* the development of a widely applicable model of environment — and related infrastructure, tools and methodologies —, *(ii)* the study of theories, formal models and languages to integrate environment and strong agency, *(iii)* the extension of mainstream platforms, frameworks and programming languages to include useful environment abstractions, and finally *(iv)* the use of environment as driving force in emerging and new applications of multiagent systems, such as simulation, self-organising systems, and bioinformatics, to mention some.