Architecture-Centric Software Development of Situated Multiagent Systems

Danny Weyns and Tom Holvoet

DistriNet, Katholieke Universiteit Leuven
Celestijnenlaan 200 A, B-3001 Leuven, Belgium
{danny.weyns,tom.holvoet}@cs.kuleuven.be

Abstract. A multiagent system (MAS) structures a software system as a set of autonomous agents that interact through a shared environment. Software architecture is generally considered as the structures of a system which comprise software elements and the relationships among the elements. So there is a clear connection between MAS and software architecture. In our research, we study situated MAS, i.e. systems in which agents have an explicit position in the environment. We apply situated MAS to domains that are characterized by highly dynamic operating conditions and an inherent distribution of resources. We use an architecture-centric approach for developing such MAS. From our experiences with building various applications, we have developed a reference architecture for situated MAS. The reference architecture provides an asset base architects can draw from when developing new systems that share the common base of the reference architecture. In this paper, we explain our perspective on architecture-centric software development of MAS. We give an overview of the reference architecture and we show an excerpt of the software architecture of an industrial application in which we have used the reference architecture. The reference architecture shows how knowledge and experience with MAS can be documented and matured in a form that has proven its value in mainstream software engineering. We believe that this integration is a key to industrial adoption of MAS.

1 Introduction

Five years of application-driven research taught us that there is a close connection between multiagent systems (MAS) and software architecture. Our perspective on the essential purpose of MAS is as follows:

A multiagent system provides the software to solve a problem by structuring the system as a number of interacting autonomous entities (agents) embedded in an environment in order to achieve the functional and quality requirements of the system.

This perspective states that a MAS provides the software to *solve* a problem. In particular, a MAS *structures the system* as a number of interacting agents embedded in an environment. The purpose of the system is to achieve the *requirements* of the system. This is exactly what *software architecture* is about. [6] defines software architecture as: “the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.” Software elements (or in general architectural elements) provide the functionality of the system, while the required quality attributes are primarily achieved through the structures of the software architecture.

As such, MAS are in essence a family—yet a large family—of software architectures. Based on the problem analysis that yields the functional and quality attribute requirements of the system, the architect may or may not choose for a MAS-based solution. Quality attribute requirements such as flexibility, openness, and robustness may be arguments for the designer to choose for a MAS software architecture. As such, we consider MAS as one valuable family of approaches to solve software problems in a large spectrum of possible ways to solve problems. Typical architectural elements of MAS software architectures are agents, coordination infrastructure, resources, services, etc. The relationships between the elements are very diverse, ranging from environment mediated interaction between cooperative agents via digital pheromone trails to complex negotiation protocols in a society of self-interested agents. In short, MAS are a rich family of architectural approaches with specific characteristics, useful for a diversity of challenging application domains. By considering MAS essentially as software architecture, MAS gets a clear and prominent role in the software development process paving a way to integrate MAS with mainstream software engineering.

Architecture-Centric Software Development of Situated MAS. In our research, we study situated MAS, i.e. systems in which agents have an explicit position in the environment. We apply situated MAS to domains that are characterized by highly dynamic operating conditions and an inherent distribution of resources. We use an architecture-centric approach for developing such MAS. From our experiences with building various applications, we have developed a reference architecture for situated MAS. The reference architecture provides an asset base architects can draw from when developing new systems that share the common base of the reference architecture. In this paper, we explain our perspective on architecture-centric software development of MAS. We give an overview of the reference architecture for situated MAS and we show an excerpt of the software architecture of an industrial application in which we have used the reference architecture.

Overview. The paper is structured as follows. We start with a brief introduction of architecture-centric software development in general. Next, in Sect. 3 we give a high-level overview of the reference architecture for situated MAS. Section 4 shows an excerpt of the software architecture of an industrial AGV transportation system in which we have used the reference architecture for architectural design. Section 5 discusses related work, and in Sect. 6 we draw conclusions.

2 Architecture-Centric Software Development

To understand our perspective on software engineering of MAS, we give a brief overview of architecture-centric software development in general. We use the evolutionary delivering life cycle [26,6], see Fig. 1. This life cycle model situates architectural design in the centre of the development activities. The main idea of the model is to support incremental software development and to incorporate early feedback from the stakeholders. The life-cycle consists of two main phases: developing the core system and delivering the final software product.

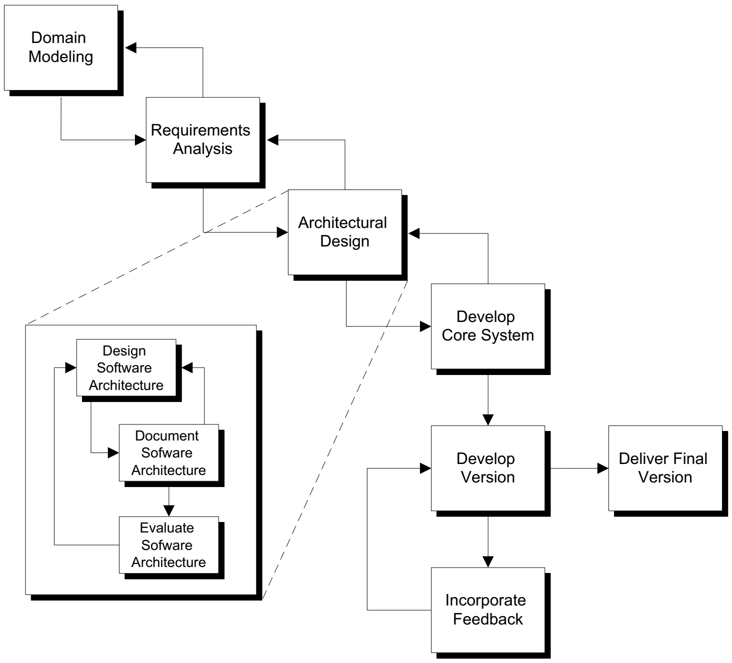

Fig. 1. Architectural design in the software development life cycle

In the first phase the core system is developed. This phase includes four activities: defining a domain model, performing a system requirements analysis, designing the software architecture, and developing the core system. Requirements analysis includes the formulation of functional requirements of the system as well as eliciting and prioritizing of the quality attributes requirements. Designing the software architecture includes the design and documentation of the software architecture, and an evaluation of the architecture. The development of the core system includes detailed design, implementation and testing. The software engineering process is an iterative process, the core system is developed incrementally, passing multiple times through the different stages of the development process. Fig. 1 shows how architectural design iterates with requirements analysis on the one hand, and with the development of the core system on the other hand. The output of the first phase is a domain model, a list of system requirements, a software architecture, and an implementation of the core of the software system.

In the second phase, subsequent versions of the system are developed until the final software product can be delivered. In principle there is no feedback loop from the second to the first phase although in practice specific architectural refinements may be necessary.

We now briefly look at architectural design and the activities it directly iterates with: requirements analysis and developing the core system.

Requirements Analysis. Gathering system requirements includes the elicitation of functional requirements as well as eliciting and prioritizing of the quality attributes requirements. Functional requirements of a system are typically expressed as use cases [25]. A use case lists the steps, necessary to accomplish a functional goal for an actor that uses the system. In our research, we also use scenarios that describe interactions among parts in the system—rather than interactions that are initiated by an external actor. An example is a scenario that describes the requirement of collision avoidance of automatic guided vehicles on crossroads. For the expression of quality requirements we use system-specific *quality attribute scenarios* [5]. A quality attribute scenario consists of three parts: (1) a stimulus: an internally or externally generated condition that affects (a part of) the system and that needs to be considered when it arrives at the system; (2) a context: the conditions under which the stimulus occurs; (3) a response: the activity that is undertaken—through the architecture—when the stimulus arrives. The response should be measurable so that the requirement can be tested. Here is an example of a quality attribute scenario:

> *An Automatic Guided Vehicle (AGV) gets broken and blocks a path under normal system operation. Other AGVs have to record this, choose an alternative route—if available—and continue their work.*

The stimulus in this example is "An Automatic Guided Vehicle gets broken and blocks a path", the context is "under normal system operation", and the response is "other AGVs have to record this, choose an alternative route—if available—and continue their work". Quality attribute scenarios provide a means to transform vaguely formulated qualities such as "the system shall be modifiable" or "the system shall exhibit acceptable flexibility" into concrete expressions. To elicit and prioritize quality attribute scenarios, we use *utility trees* [14]. An utility tree compels the architect and other stakeholders involved in a system to define the relevant quality requirements precisely. An utility tree consists of four levels. The root node of the tree is *utility* expressing the overall quality of the system. High-level quality attributes form the second level of the tree. Each quality attribute is further refined in the third level. Finally, the leaf nodes of the tree are the quality attribute scenarios. Eah scenario is assigned a ranking that expresses its priority relatively to the other scenarios. Criteria for prioritization include the importance of the scenario to the success of the system, and the difficulty to achieve the scenario. It is clear that the most important scenarios are those that have a high ranking on both criteria. [8] shows an example of a utility tree for the automatic transportation system we discuss in section 4.

Architectural Design. Architectural design includes the design and documentation of the software architecture, and an evaluation of the architecture (see Fig. 1).

Design. Designing a software architecture is moving from system requirements to architectural decisions. The various requirements are achieved by architectural decisions that are based on architectural approaches. One common architectural approach are architectural patters [31]. An architectural pattern is a description of architectural elements and their relationships that has proven to be useful for achieving particular qualities. Examples of architectural patterns are layers and blackboard. In our research, we have developed a reference architecture for MAS as a reusable architectural approach. This reference architecture integrates a set of architectural patterns that have proven their value in various MAS applications we have studied and built. A reference architecture provides an integrated set of architectural patterns the architect can draw from to select suitable architectural solutions.

Architectural design requires a systematic approach to develop a software architecture that meets the required functionality and satisfies the quality requirements. In our research, we use techniques from the Attribute Driven Design (ADD [10,6]) method to design the architecture for a software system with a reference architecture. ADD is a decomposition method that is based on understanding how to achieve quality goals through proven architectural approaches. Usually, the architect starts from the system as a whole and then iteratively refines the architectural elements, until the elements are sufficiently fine-grained to start detailed design and implementation. At that point, the software architecture becomes a prescriptive plan for construction of the system that enables effective satisfaction of the systems functional and quality requirements [21,13].

A reference architecture serves as a blueprint to *guide* the architect through the decomposition process. In particular, the ADD process can be used to iteratively refine the software architecture, and the reference architecture can serve as a guidance in this decomposition process. In addition, common architectural approaches have to be applied to refine and extend architectural elements when necessary according to the requirements of the system at hand.

Documentation. A software architecture is described by different *views*. Each view belongs to a *viewtype* [13]. A viewtype defines the elements and relationship used to describe the architecture of a software system from a particular perspective. We use three different viewtypes:

1. The module viewtype: views in this viewtype document a system's principal units of implementation.
2. The component-and-connector viewtype: views in this viewtype document the system's units of execution.

3. The deployment viewtype: views in this viewtype document the relationships between a system's software and its development and execution environment.

Documenting a software architecture comes down to documenting the relevant views of the software architecture for the application at hand. Each view is documented by means of a number of view packets [13]. A view packet is a small, relatively self-contained bundle of information of the reference architecture.

Evaluation. A software architecture is the foundation of a software system, it represents a system's earliest set of design decisions. Due to its large impact on the development of the system, it is important to verify the architecture as soon as possible. Modifications in early stages of the design are cheap and easy to carry out. Deferring evaluation might require expensive changes or even result in a system of inferior quality.

The evaluation of software architecture is an active research topic, see e.g. [4,27]. In our research, we use the Architectural Tradeoff Analysis Method [14] (ATAM). ATAM is a well-established method for software architecture evaluation developed at the Software Engineering Institute [2]. The ATAM incites the stakeholders to articulate specific quality goals and to prioritize conflicting goals; it forces the architect to provide a clear explanation and documentation of the software architecture; and especially it uncovers problems with the architecture that can be used to improve the quality of the software architecture in an early stage of the development cycle. An ATAM evaluation produces the following results:

- A prioritized list of quality attribute requirements in the form of a quality attribute utility tree.
- A mapping of architectural approaches to quality attributes. The analysis of the architecture exposes how the architecture achieves—or fails to achieve—the important quality attribute requirements.
- Risks and non-risks. Risks are potentially problematic architectural decisions, non-risks are good architectural decisions.
- Sensitivity points and tradeoff points. A sensitivity point is an architectural decision that is critical for achieving a particular quality attribute. A tradeoff point is an architectural decision that affects more than one attribute, it is a sensitivity point for more than one attribute.

[9] discusses our experiences with ATAM for the application discussed in Sect. 4.

Developing the Core System. The development of the core system includes detailed design, implementation and testing. The software architecture defines constraints on detailed design and implementation, it describes how the implementation must be divided into elements and how these elements must interact with one another to fulfil the system requirements. On the other hand, a software architecture does not *define* an implementation, many fine-grained design decisions are left open by the architecture and must be completed by designers and developers. For some tasks established techniques can be used such as design patterns or well-know algorithms. However, other—MAS specific—tasks require dedicated design guidelines, e.g. the detailed design of an agent communication language or a pheromone infrastructure.

3 Reference Architecture for Situated Multiagent Systems

In our research, we study the engineering of software systems with the following main characteristics and requirements:

- Stakeholders of the systems (users, project managers, architects, developers, maintenance engineers, etc.) have various—often conflicting—demands on the quality of the software. Important quality requirements are flexibility (adapt to variable operating conditions) and openness (cope with parts that come and go during execution).
- The software systems are subject to highly dynamic and changing operating conditions, such as dynamically changing workloads and variations in availability of resources and services. An important requirement of the software systems is to manage the dynamic and changing operating conditions autonomously.
- Global control is hard to achieve. Activity in the systems is inherently localized, i.e. global access to resources is difficult to achieve or even infeasible. The software systems are required to deal with the inherent locality of activity.

Example domains are mobile and ad-hoc networks, sensor networks, automated transportation and traffic control systems, and manufacturing control.

To deal with these requirement we apply the paradigm of situated MAS. During the last five years, we have developed several mechanisms of adaptivity for situated MAS, including selective perception [46], protocol-based communication [45], behavior-based decision making with roles and situated commitments [33], and laws that mediate the activities of agents in the environment [38]. We have applied these mechanisms in various applications, ranging from experimental simulations [37,39,36] and prototypical robot applications [44,33] up to an industrial transportation system for automatic guided vehicles [43,40,9].

Based on these experiences, we have developed a reference architecture for situated MAS. Motivations for the reference architecture are: (1) it integrates the different mechanisms for adaptivity. It defines how the functionalities of the various mechanisms are allocated to software elements of agents and the environment and how these elements interact with one another, (2) it provides a reusable design artifact, the reference architecture facilitates deriving new software architectures for systems that share the common base more reliably and cost effectively, and (3) the reference architecture embodies the knowledge and expertise we have acquired during our research. It conscientiously documents the know-how obtained from this research. As such, the reference architecture offers a vehicle to study and learn the advanced perspective on situated MAS we have developed.

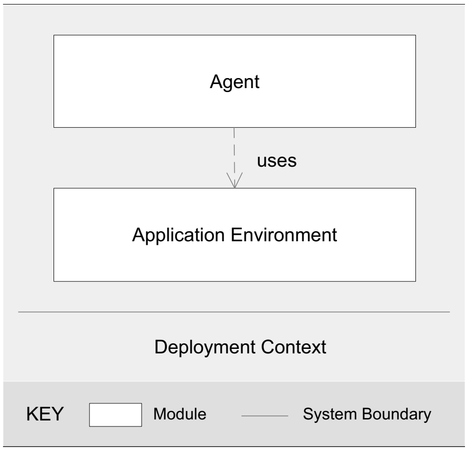

Fig. 2. Top-level module decomposition of a situated MAS

Fig. 2 shows the top-level module decomposition of the reference architecture of situated MAS that shows the main software units in the system.

A situated multiagent system is decomposed in two basic modules: `Agent` and `Application Environment`.

`Agent` is an autonomous problem solving entity in the system. An agent encapsulates its state and controls its behavior. The responsibility of an agent is to achieve its design objectives, i.e. to realize the application specific goals it is assigned. Agents are situated in an environment which they can perceive and in which they can act and interact with one another. Agents are able to adapt their behavior according to the changing circumstances in the environment. A situated agent is a cooperative entity. The overall application goals result from interaction among agents, rather than from sophisticated capabilities of individual agents.

A concrete MAS application typically consists of agents of different agent types. Agents of different agent types typically have different capabilities and are assigned different application goals.

The `Application Environment` is the part of the environment that has to be designed for a concrete MAS application. The application environment enables agents to share information and to coordinate their behavior. The core responsibilities of the application environment are:

- To provide access to external entities and resources.
- To enable agents to perceive and manipulate their neighborhood, and to interact with one another.
- To mediate the activities of agents. As a mediator, the environment not only enables perception, action and interaction, it also constrains them.

The application environment provides functionality to agents on top of the *deployment context*. The deployment context consists of the given hardware and software and external resources such as sensors and actuators, a printer, a network, a database, a web service, etc.

As an illustration, a peer-to-peer file sharing system is deployed on top of a deployment context that consists of a network of nodes with files and possibly other resources. The application environment enables agents to access the external resources, shielding low-level details. Additionally, the application environment may provide a coordination infrastructure that enables agents to coordinate their behavior. E.g., the application environment of a peer-to-peer file share system can offer a pheromone infrastructure to agents that they can use to dynamically form paths to locations of interest.

Thus, we consider the *environment* as consisting of two parts, the deployment context and the application environment [42]. The internal structure of the deployment context is not considered in the reference architecture. For a distributed application, the deployment context consists of multiple processors deployed on different nodes that are connected through a network. Each node provides an application environment to the agents located at that node. Depending on the specific application requirements, different application environment types may be provided. For some applications, the same type of application environment subsystem is instantiated on each node. For other applications, specific types are instantiated on different nodes, e.g., when different types of agents are deployed on different nodes.

In the next section, we zoom in on the collaborating components view of the reference architecture. For a description of other architectural views of the reference architecture and a formal specification of the various architectural elements we refer to [34].

3.1 Collaborating Components View Packets

The collaborating components view shows the MAS or parts of it as a set of interacting runtime components that use a set of shared data repositories to realize the required system functionalities. The elements of the collaborating components view are:

- *Runtime components.* Runtime components achieve a part of the system functionality. Runtime components are instances of modules described in the module decomposition view.
- *Data repositories.* Data repositories enable multiple runtime components to share data. Data repositories correspond to the shared data repositories described in the component and connector shared data view.
- *Component–repository connectors.* Component–repository connectors connect runtime components which data repositories. These connectors determine which runtime components are able to read and write data in the various data repositories of the system.
- *Component–component connectors.* Collaborating components require functionality from one another and provide functionality to one another.

Component–component connectors enable runtime components to request each other to perform a particular functionality.

The collaborating components view is an excellent vehicle to learn the runtime behavior of a situated MAS. The view shows the data flows between runtime components and the interaction with data stores, and it specifies the functionalities of the various components in terms of incoming and outgoing data flows.

We discuss two view packets of the collaborating components view. We start with the view packet that describes the collaborating components of agent. Next, we discuss the view packet that describes the collaborating components of the application environment.

A. Collaborating Components View Packet: Agent

Primary Presentation. The primary presentation is show in Fig. 3.

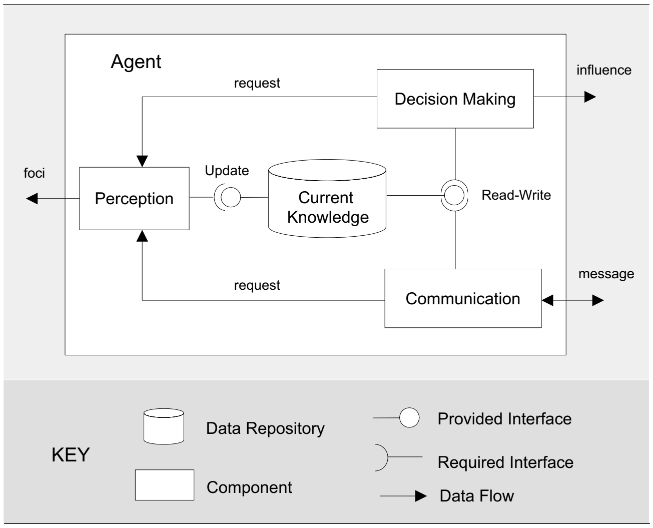

Fig. 3. Collaborating Components of Agent

`Elements and their Properties.` The `Agent` component (i.e. a runtime instance of the Agent module shown in Fig. 2) consists of three subcomponents: `Perception`, `Decision Making`, and `Communication`. These components share the `Current Knowledge` repository. We first give a brief explantion of the responsibilities of the components and then we explain the collaboration between the components and the shared data repository.

`Perception` is responsible for collecting runtime information from the environment (application environment and deployment context). The perception component supports selective perception [46]. Selective perception enables an agent to direct its perception according to its current tasks. To direct its perception an agent selects a set of foci and filters. Foci allow the agent to sense the environment only for specific types of information. Sensing results in a representation of the sensed environment. A representation is a data structure that represents elements or resources in the environment. The perception module maps this representation to a percept, i.e. a description of the sensed environment in a form of data elements that can be used to update the agent's current knowledge. The selected set of filters further reduces the percept according to the criteria specified by the filters.

`Decision Making` is responsible for action selection. The action model of the reference architecture is based on the influence–reaction model introduced in [15]. This action model distinguishes between influences that are produced by agents and are attempts to modify the course of events in the environment, and reactions, which result in state changes in the environment. The responsibility of the decision making module is to select influences to realize the agent's tasks, and to invoke the influences in the environment [38].

Situated agents use a behavior-based action selection mechanism [41]. To enable situated agents to set up collaborations, we have extended behavior-based action selection mechanisms with roles and situated commitments [44,33,45]. A role represents a coherent part of an agent's functionality in the context of an organization. A situated commitment is an engagement of an agent to give preference to the actions of a particular role in the commitment. Agents typically commit relative to one another in a collaboration, but an agent can also commit to itself, e.g. when a vital task must be completed. Roles and commitments have a well-known *name* that is part of the domain ontology and that is shared among the agents in the system. Sharing these names enable agents to set up collaborations via message exchange. We explain the coordination among decision making and communication in the design rationale of this view packet.

`Communication` is responsible for communicative interactions with other agents. Message exchange enables agents to share information and to set up collaborations. The communication module processes incoming messages, and produces outgoing messages according to well-defined communication protocols [45]. A communication protocol specifies a set of possible sequences of messages. We use the notion of a *conversation* to refer to an ongoing communicative interaction. A conversation is initiated by the initial message of a communication protocol. At each stage in the conversation there is a limited set of possible messages that can be exchanged. Terminal states determine when the conversation comes to an end.

The information exchanged via a message is encoded according to a shared communication language. The communication language defines the format of the messages, i.e. the subsequent fields the message is composed of. A message includes a field with a unique identifier of the ongoing conversation to which the message belong, fields with the identity of the sender and the identities of the addressees of the message, a field with the performative of the message, and a field with the content of the message. Communicative interactions among agents are based on an *ontology* that defines a shared vocabulary of words that agents use in messages. The ontology enables agents to refer unambiguously to concepts and relationships between concepts in the domain when exchanging messages.

`Current Knowledge` repository contains data that is shared among the data accessors. Data stored in the current knowledge repository refers to state perceived in the environment, to state related to the agent's roles and situated commitments, and possibly other internal state that is shared among the data accessors. Fig. 3 shows the interconnections between the current knowledge repository and the internal components of the agent. These interconnections are called assembly connectors [3]. An assembly connector ties one component's provided interface with one or more components' required interfaces, and is drawn as a lollipop and socket symbols next to each other. Provided and required interfaces per assembly connector share the same name.

The current knowledge repository exposes two interfaces. The provided interface `Update` enables the perception component to update the agents knowledge according to the information derived from sensing the environment. The `Read-Write` interface enables the communication and decision making component to access and modify the agent's current knowledge.

Collaborations. The overall behavior of the agent is the result of the coordination of two components: decision making and communication. Decision making is responsible for selecting suitable influences to act in the environment. Communication is responsible for the communicative interactions with other agents. When selecting actions and communicating messages with other agents, decision making and communication typically request perceptions to update the agent's knowledge about the environment. By selecting an appropriate set of foci and filters, the agent directs its attention to the current aspects of its interest, and adapts it attention when the operating conditions change.

To complete the agent's tasks, decision making and communication coordinate via the current knowledge repository. For example, agents can send each other messages with requests for information that enable them to act more purposefully. Decision making and communication also coordinate during the progress of a collaboration. Collaborations are typically established via message exchange. Once a collaboration is achieved, the communication module activates a situated commitment. This commitment will affect the agent's decision making towards actions in the agent's role in the collaboration. This continues until the commitment is deactivated and the collaboration ends.

The separation of functionality for coordination (via communication) from the functionality to perform actions to complete tasks has several advantages, including clear design, improved modifiability and reusability. Two particular advantages are: (1) it allows both functions to act in parallel, and (2) it allows both functions to act at a different pace. In many applications, sending messages and executing actions happen at different tempo; a typical example is robotics. Separation of communication from performing actions enables agents to reconsider the coordination of their behavior while they perform actions, improving adaptability and efficiency.

B. Collaborating Components View Packet: Application Environment

Primary Presentation. The primary presentation is show in Fig. 4.

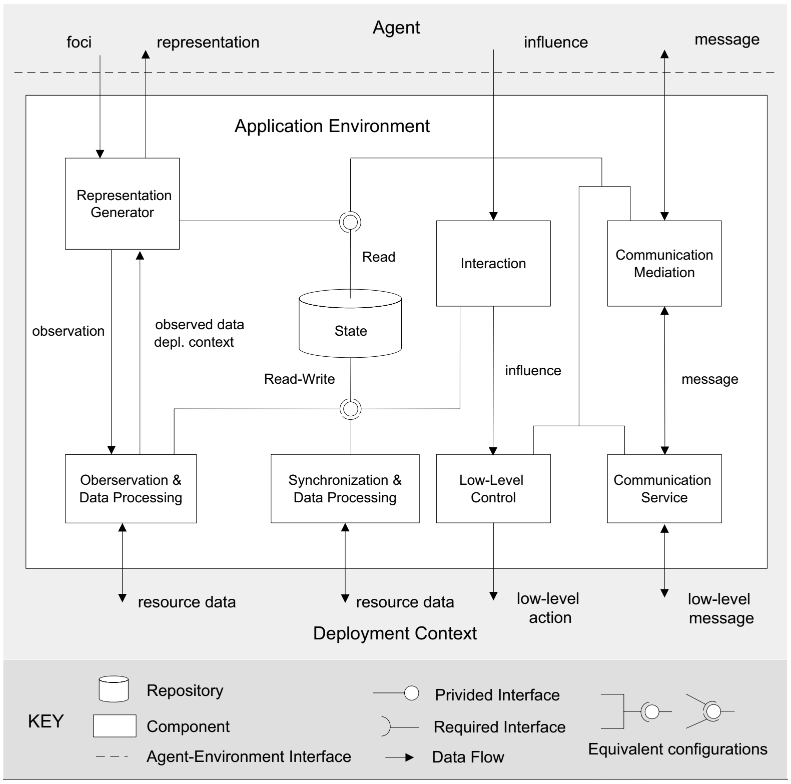

Fig. 4. Collaborating Components of Application Environment

Elements and their Properties The `Application Environment` component consists of seven subcomponents and the shared `State` repository. We discuss the responsibilities of each of the elements in turn. Then, we zoom on the collaboration between de components.

The **State** repository contains data that is shared between the components of the application environment. Data stored in the state repository typically includes an abstraction of the deployment context together with additional state related to the application environment. Examples of state related to the deployment context are a representation of the local topology of a network, and data derived from a set of sensors. Examples of additional state are the representation of digital pheromones that are deployed on top of a network, and virtual marks situated on the map of the physical environment. The state repository may also include agent-specific data, such as the agents' identities, the positions of the agents, and tags used for coordination purposes.

The `Representation Generator` provides the functionality to agents for perceiving the environment. When an agent senses the environment, the representation generator uses the current state of the application environment and possibly state collected from the deployment context to produce a representation for the agent. Agents' perception is subject to perception laws that provide a means to constrain perception [46]. For example, for reasons of efficiency a designer can introduce default limits for perception in order to restrain the amount of information that has to be processed, or to limit the occupied bandwidth.

`Observation & Data Processing` provides the functionality to observe the deployment context and collect date from other nodes in a distributed setting. The observation & data processing module translates observation requests into observation primitives that can be used to collect the requested data from the deployment context. Data may be collected from external resources in the deployment context or from the application environment instances on other nodes in a distributed application. The observation & data processing module can provide additional functions to pre-process data, examples are sorting and integration of observed data.

`Interaction` is responsible to deal with agents' influences in the environment. Agents' influences can be divided in two classes: influences that attempt to modify state of the application environment and influences that attempt to modify the state of resources of the deployment context. An example of the former is an agent that drops a digital pheromone in the environment. An example of the latter is an agent that writes data in an external data base. Agents' influences are subject to action laws [38]. Action laws put restrictions on the influences invoked by the agents, representing domain specific constraints on agents' actions. For example, when several agents aim to access an external resource simultaneously, an interaction law may impose a policy on the access of that resource. For influences that relate to the application environment, the interaction module calculates the reaction of the influences resulting in an update of the state of the application environment. Influences related to the deployment context are passed to the Low-Level Control module.

`Low-Level Control` bridges the gap between influences used by agents and the corresponding action primitives of the deployment context. Low-level control converts the influences invoked by the agents into low-level action primitives in the deployment context. This decouples the interaction module from the details of the deployment context.

The `Communication Mediation` mediates the communicative interactions among agents. It is responsible for collecting messages; it provides the necessary infrastructure to buffer messages, and it delivers messages to the appropriate agents. Communication mediation regulates the exchange of messages between agents according a set of applicable communication laws [45]. Communication laws impose constraints on the message stream or enforce domain–specific rules to the exchange of messages. Examples are a law that drops messages directed to agents outside the communication–range of the sender and a law that gives preferential treatment to high-priority messages. To actually transmit the messages, communication mediation makes use of the Communication Service module.

`Communication Service` provides that actual infrastructure to transmit messages. Communication service transfers message descriptions used by agents to communication primitives of the deployment context. For example, FIPA ACL message [16] enable a designer to express the communicative interactions between agents independently of the applied communication technology. However, to actually transmit such messages, they have to be translated into low-level primitives of a communication infrastructure provided by the deployment context. Depending on the specific application requirements, the communication service may provide specific communication services to enable the exchange of messages in a distributed setting, such as white and yellow page services. An example infrastructure for distributed communication is Jade [7]. Specific middleware may provide support for communicative interaction in mobile and ad-hoc network environments, an example is discussed in [30].

`Synchronization & Data Processing` synchronizes state of the application environment with state of resources in the deployment context as well as state of the application environment on different nodes. State updates may relate to dynamics in the deployment context and dynamics of state in the application environment that happens independently of agents or the deployment context. An example of the former is the topology of a dynamic network which changes are reflected in a network abstraction maintained in the state of the application environment. An example of the latter is the evaporation of digital pheromones. Middleware may provide support to collect data in a distributed setting. An example of middleware support for data collection in mobile and ad-hoc network environments is discussed in [29]. Synchronization & data processing converts the resource data observed from the deployment context into a format that can be used to update the state of the application environment. Such conversion typically includes a processing or integration of collected resource data.

Collaborations. Successively, we zoom in on the collaborating components for perception, interaction, communication, and the synchronization of state among nodes and with resources in the deployment context.

Perception. The representation generator collects perception requests from the agents and generates representations according to the given foci. Representation generator collects the required state from the state repository, and optionally it requests observation & data processing to collect additional data from the deployment context and possibly state of other nodes. State collection is subject to the perception laws. Observation & data processing returns the observed data to representation generator that generates a representation that is returned to the requesting agent.

Interaction. Interaction collects the concurrently invoked influences of agents and converts them into operations. The execution of operations is subject to the action laws of the system. Operations that attempt to modify state of the application environment are immediately executed by the interaction component. Operations that attempt to modify state of the deployment context are forwarded to low-level control that converts the operations into low–level interactions in the deployment context.

Communication. Communication mediation handles the communicative interactions among agents. The component collects the messages sent by agents, applies the communication laws, and subsequently passes the messages to the communication service. This latter component converts the messages directed to agents on other nodes into low–level interactions that are transmitted via the deployment context. Furthermore, communication service collects low–level messages from the deployment context, converts the messages into a format understandable for the agents, and forward the messages to communication mediation that delivers the messages to the appropriate agents. Messages directed to agents that are located at the same node are directly transferred to the appropriate agents.

State Synchronization. Synchronization & data processing performs its tasks independently of other components of the application environment. To synchronize the state of the application environment in a distributed setting, synchronization & data processing components on different nodes have to coordinate according to the requirements of the application at hand.

4 Excerpt of a Software Architecture for an AGV Transportation System

We now illustrate how we have used the reference architecture for the architectural design of an automated transportation system for warehouse logistics that has been developed in a joint R&D project between the DistriNet research group and Egemin, a manufacturer of automating logistics services in warehouses and manufactories [43,1]. The transportation system uses automatic guided vehicles (AGVs) to transport loads through a warehouse. Typical applications include distributing incoming goods to various branches, and distributing manufactured products to storage locations. AGVs are battery-powered vehicles that can navigate through a warehouse following predefined paths on the factory floor. The low-level control of the AGVs in terms of sensors and actuators such as staying on track on a path, turning, and determining the current position is handled by the AGV control software.

4.1 Multiagent System for the AGV Transportation System

In the project, we have applied a MAS approach for the development of the transportation system. The transportation system consists of two kinds of agents: transport agents and AGV agents. Transport agents represent tasks that need to be handled by an AGV and are located at a transport base, i.e. a stationary computer system. AGV agents are responsible for executing transports and are located in mobile vehicles. The communication infrastructure provides a wireless network that enables AGV agents at vehicles to communicate with each other and with transport agents on the transport base.

AGVs are situated in a physical environment, however this environment is very constrained: AGVs cannot manipulate the environment, except by picking and dropping loads. This restricts how AGV agents can exploit their environment. Therefore, a virtual environment was introduced for agents to inhabit. This virtual environment provides an interaction medium that agents can use to exchange information and coordinate their behavior. The virtual environment is necessarily distributed over the AGVs and the transport base, i.e. a local virtual environment is deployed on each AGV and the transport base. The local virtual environment corresponds to the application environment in the reference architecture. State on local virtual environments is merged opportunistically, as the need arises. The synchronization of the state of neighboring local virtual environments is supported by the ObjectPlaces middleware [29,30]. The AGV control system is developed on top of the .NET framework and programmed in C#.

As an illustration of the software architecture of the AGV transportation system, we zoom on the collaborating components view of the local virtual environment that is deployed on the AGVs.

4.2 Collaborating Components View of the Local Virtual Environment

Fig. 5 shows the collaborating components view of the local virtual environment.

The general structure of the local virtual environment is related to the structure of the application environment in the reference architecture as follows. The state repository corresponds to the state repository in the reference architecture, see Fig. 4 in section 3.1. The state elements are specific to the local virtual environment of an AGV control system. The perception manager provides the functionality for selective perception of the environment, similar to the representation generator in the reference architecture. Contrary to the representation generator, the perception manager interacts only with the state repository; the functionality of the observation & data processing component in the reference architecture is absent in the local virtual environment. The action manager corresponds to the interaction component of the application environment. Low-level control corresponds with E'nsor, i.e. the control software to interact with the sensors and actuators of the AGV. We fully reused E'nsor in the project. The communication manager integrates the responsibilities of communication mediation and the communication service of the application environment. The communication service handles the bidirectional translation of messages and manages message transmission via .Net remoting. Finally, the laws for perception, action, and communication, are integrated in the applicable components.

Elements and Their Properties

`State.` Since the virtual environment is necessarily distributed over the AGVs and the transport base, each local virtual environment is responsible to keep its

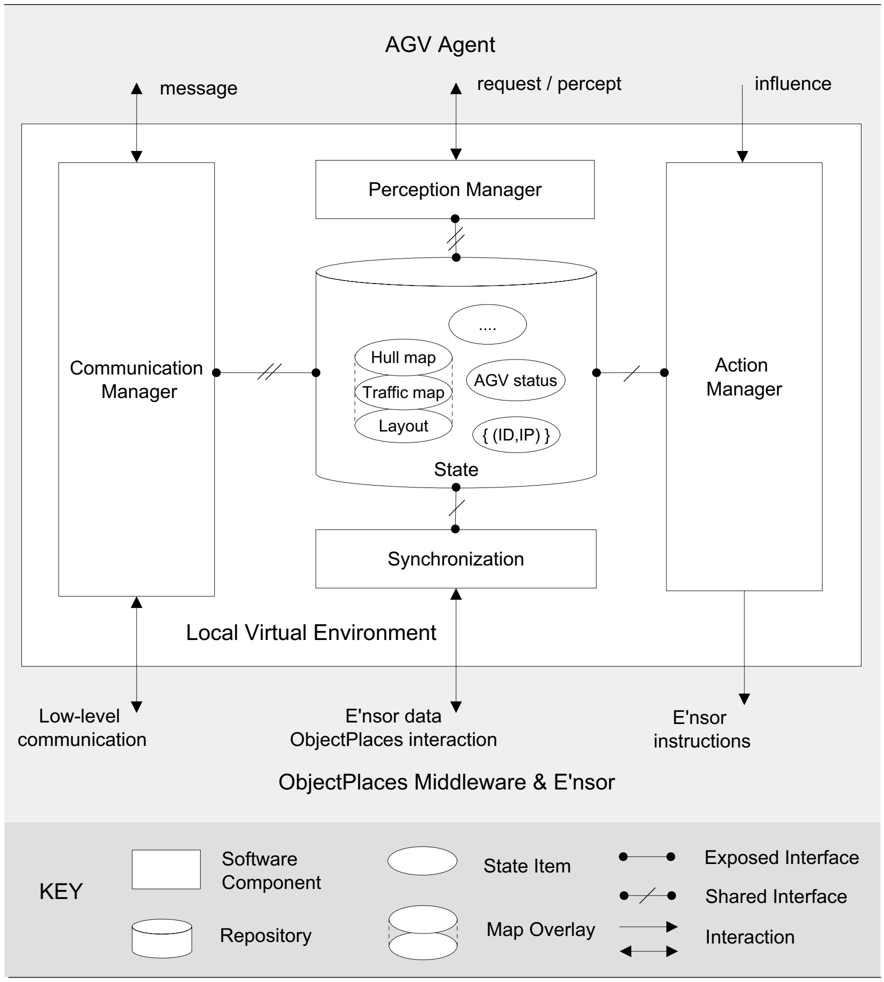

Fig. 5. Collaborating components view of the local virtual environment of AGVs

state synchronized with other local virtual environments. The state of the local virtual environment is divided into three categories:

1. Static state: this is state that does not change over time. Examples are the layout of the factory floor, which is needed for the AGV agent to navigate, and `(AGV id, IP number)` tuples used for communication. Static state must never be exchanged between local virtual environments since it is common knowledge and never changes.
2. Observable state: this is state that can be changed in one local virtual environment, while other local virtual environments can only observe the state. An AGV obtains this kind of state from its sensors directly. An example is an AGV's position. Local virtual environments are able to observe another

AGV's position, but only the local virtual environment on the AGV itself is able to read it from its sensor, and change the representation of the position in the local virtual environment. No conflict arises between two local virtual environments concerning the update of observable state.
3. Shared state: this is state that can be modified in two local virtual environments concurrently. An example is a hull map. AGV agents mark the path they are going to drive in their local virtual environment using hulls. The hull of an AGV is the physical area the AGV occupies. A series of hulls describe the physical area an AGV occupies along a certain path. AGV agents use hull for collision avoidance. When the local virtual environments on different machines synchronize, the local virtual environments must generate a consistent and up-to-date state in both local virtual environments.

`Perception Manager` handles perception in the local virtual environment. The perception manager's task is straightforward: when the agent requests a percept, for example the current positions of neighboring AGVs, the perception manager queries the necessary information from the state repository of the local virtual environment and returns the percept to the agent. Perception is subject to laws that restrict agents perception of the virtual environment. For example, when an agent senses the hulls for collision avoidance of neighboring AGVs, only the hulls within collision range are returned to the AGV agent.

`Action Manager` handles agents' influences. AGV agents can perform two kinds of influences. One kind of influences are commands to the AGV, for example moving over a segment and picking up a load. These influences are handled fairly easily by translating them and passing them to the E'nsor control software. A second kind of influences attempt to manipulate the state of the local virtual environment. Putting marks in the local virtual environment is an example. An influence that changes the state of the local virtual environment may in turn trigger state changes of neighboring local virtual environments (see Synchronization below). Influences are subject to laws, e.g., when an AGV agent projects a hull in the local virtual environment, this latter determines when an AGV acquires the right to move on. In particular, if the area is not marked by other hulls (the AGV's own hulls do not intersect with others), the AGV can move along and actually drive over the reserved path. In case of a conflict, the involved local virtual environments use the priorities of the transported loads and the vehicles to determine which AGV can move on. AGV agents monitor the local virtual environment and only instruct the AGV to move on when they are allowed. Afterwards, the AGV agents remove the markings in the environment. This example shows that the local virtual environment serves as a flexible coordination medium: agents coordinate by putting marks in the environment, and observing marks from other agents.

`Communication Manager` is responsible for exchanging messages between agents. Agents can communicate with other agents through the virtual environment. A typical example is an AGV agent that communicates with a transport agent to assign a transport. Another example is an AGV agent that requests the AGV agent of a waiting AGV to move out of the way. The communication manager translates the high-level messages to low-level communication instructions that can be sent through the network and vise versa (resolving agent names to IP numbers, etc.). Communication is subject to laws, an example is the restriction of communication range for messages used for transport assignment [35].

`Synchronization` has a dual responsibility. It periodically polls E'nsor and updates the state of the local virtual environment accordingly. An example is the maintenance of the actual position of the AGV in the local virtual environment. Furthermore, synchronization is responsible for synchronizing state between local virtual environments of neighboring machines. An example is the synchronization of hulls on neighboring AGVs.

Design Rationale

Changes in the system (e.g., AGVs that enter/leave the system) are reflected in the state of the local virtual environment, releasing agents from the burden of such dynamics. As such, the local virtual environment—supported by the ObjectPlaces middleware—supports openness.

Since an AGV agent continuously needs up-to-date data about the system (position of the vehicles, status of the battery, etc.), we decided to keep the representation of the relevant state of the deployment context in the local virtual environment synchronized with the actual state. Therefore, E'nsor and the ObjectPlaces middleware are periodically polled to update the status of the system. As such, the state repository maintains an accurate representation of the state of the system to the AGV agent.

5 Related Work

Current practice in agent-oriented software engineering considers MAS as a radically new way of engineering software. For example, in [10], Wooldridge et al. state "There is a fundamental mismatch between the concepts used by object-oriented developers and other mainstream software engineering paradigms, and the agent-oriented view. [...] Existing software development techniques are unsuitable to realize the potential of agents as a software engineering paradigm." As a result, numerous MAS methodologies have been developed [19]. Although some of the methodologies adopt techniques and practices from mainstream software engineering, such as object-oriented techniques and the Unified Modeling Language, nearly all methodologies take an independent position, little or not related to mainstream software engineering practice. The position of being a radically new paradigm for software development isolates agent-oriented software engineering from mainstream software engineering. In contrast, the architecture-centric perspective on MAS we follow in our research aims to integrate MAS in mainstream software engineering.

Related work that explicitly connects MAS with software architecture is rather limited. We briefly discuss a number of representative examples. In [32], Shehory presents an initial study on the role of MAS as a software architecture style. We share the author's observation that the largest part of research in the design of MAS addresses the question: given a computational problem, can one build a MAS to solve it? However, a more fundamental question is left unanswered: given a computational problem, is a MAS an appropriate solution? An answer to this question should precede the previous one, lest MAS may be developed where much simpler, more efficient solutions apply. Almost a decade later, the majority of researchers in agent-oriented software engineering still pass over the analysis whether a MAS is an appropriate solution for a given problem.

As part of the Tropos methodology [18], a set of architectural styles were proposed which adopt concepts from organization management theory [24,12]. The styles are modelled using the $i^\star$ framework [48] which offers modelling concepts such as actor, goal, and actor dependency. Styles are evaluated with respect to various software quality attributes. The specification of quality attributes is based on te notion of softgoal. [24] states that softgoals do not have a formal definition, and are amenable to a more qualitative kind of analysis. Whereas we use a utility tree to prioritize quality requirements and to determine the drivers for architectural design, Tropos does not consider a systematic prioritization of quality goals. In Tropos, a designer visualizes the design process and simultaneously attempts to satisfy the collection of softgoals for a system.

PROSA is an acronym for Product–Resource–Order–Staff Architecture and defines a reference architecture for a family of coordination and control application, with manufacturing systems as the main domain [47]. These systems are characterized by frequent changes and disturbances. PROSA aims to provide the required flexibility to cope with these dynamics. [20] presents an interesting extension of PROSA in which the environment is exploited to obtain BDI (Believe, Desire, Intention [28]) functionality for the various PROSA agents. The PROSA reference architecture embodies architectural knowledge of a particular *problem* domain. On the contrary, the reference architecture for situated MAS embodies architectural knowledge in terms of a particular *solution* approach.

In [17], Garcia et al. observe that several concerns such as autonomy, learning, and mobility crosscut each other and the basic functionality of agents. The authors state that existing approaches that apply well-known patterns to structure agent architectures—an example is the layered architecture of Kendall [22]—fail to cleanly separate the various concerns. This results in architectures that are difficult to understand, reuse, and maintain. To cope with the problem of crosscutting concerns, the authors propose an aspect-oriented approach to structure agent architectures. An aspect-oriented agent architecture consists of a "kernel" that encapsulates the core functionality of the agent (essentially the agent's internal state), and a set of aspects [23]. Each aspect modularizes a particular concern of the agent. Yet, it is unclear whether the interaction of the different concerns in the kernel (feature interaction [11]) will not lead to similar problems the approach initially aimed to resolve. Anyway, crosscutting concerns in MAS are hardly explored and provide an interesting venue for future research.

6 Conclusions

There is a close connection between MAS and software architecture, yet, this connection is often neglected or remains implicit. In our research, we have derived a reference architecture for situated MAS from various applications we have studied and built. This reference architecture provides a blueprint to develop new software architectures for systems that have similar characteristics and requirements as the systems from which it was derived.

The reference architecture shows how knowledge and experiences with MAS can systematically be documented and maturated in a form that has proven its value in mainstream software engineering. Rather than considering MAS as a radical new way of engineering software, we believe that the integration of MAS in mainstream software engineering is a key to industrial adoption of MAS.

References

1. EMC^2: Egemin Modular Controls Concept, Project Supported by the Institute for the Promotion of Innovation Through Science and Technology in Flanders (IWTVlaanderen), (8/2006), `http://emc2.egemin.com/`
2. Software Engineering Institute: Carnegie Mellon University, (8/2006), `http://www.sei.cmu.edu/`
3. The Unified Modeling Language: (8/2006), `http://www.uml.org/`
4. Al-Naeem, T., Gorton, I., Babar, M.A., Rabhi, F., Benatallah, B.: A Quality-Driven Systematic Approach for Architecting Distributed Software Applications. In: 27th International Conference on Software Engineering, Orlando, Florida (2005)
5. Barbacci, M., Klein, M., Longstaff, T., Weinstock, C.: uality Attribute Workshops. Technical Report CMU/SEI-95-TR-21, Software Engineering Institute, Carnegie Mellon University, PA, USA (1995)
6. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison Wesley, London, UK (2003)
7. Bellifemine, F., Poggi, A., Rimassa, G.: Jade, A FIPA-compliant Agent Framework. In: 4th International Conference on Practical Application of Intelligent Agents and Multi-Agent Technology, London, UK (1999)
8. Boucké, N., Holvoet, T., Lefever, T., Sempels, R., Schelfthout, K., Weyns, D., Wielemans, J.: Applying the Architecture Tradeoff Analysis Method to an Industrial Multiagent System Application. In: Technical Report CW 431. Department of Computer Science, Katholieke Universiteit Leuven, Belgium (2005)
9. Boucké, N., Weyns, D., Schelfthout, K., Holvoet, T.: Applying the ATAM to an Architecture for Decentralized Contol of a AGV Transportation System. In: Hofmeister, C., Crnkovic, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214, pp. 180–198. Springer, Heidelberg (2006)
10. Buchmann, F., Bass, L.: Introduction to the Attribute Driven Design Method. In: 23rd International Conference on Software Engineering, Toronto, Ontario, Canada. IEEE Computer Society Press, Los Alamitos (2001)
11. Calder, M., Kolberg, M., Magill, E., Reiff-Marganiec, S.: Feature Interaction: A Critical Review and Considered Forecast. Comp. Netw. 41(1), 115–141 (2003)
12. Castro, J., Kolp, M., Mylopoulos, J.: Towards Requirements-Driven Information Systems Engineering: The Tropos Project. Inf. Syst. 27(6), 365–389 (2002)

13. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond. Addison-Wesley, London, UK (2002)
14. Clements, P., Kazman, R., Klein, M.: Evaluating Software Architectures: Methods and Case Studies. Addison Wesley, London, UK (2002)
15. Ferber, J., Muller, J.: Influences and Reaction: a Model of Situated Multiagent Systems. In: 2nd International Conference on Multi-agent Systems, Japan. AAAI Press, Stanford, California, USA (1996)
16. FIPA: Foundation for Intelligent Physical Agents, FIPA Abstract Architecture Specification (8/2006), `http://www.fipa.org/repository/bysubject.html`
17. Garcia, A., Kulesza, U., Lucena, C.: Aspectizing Multi-Agent Systems: From Architecture to Implementation. In: Choren, R., Garcia, A., Lucena, C., Romanovsky, A. (eds.) Software Engineering for Multi-Agent Systems III. LNCS, vol. 3390. Springer, Heidelberg (2005)
18. Giunchiglia, F., Mylopoulos, J., Perini, A.: The TROPOS Software Development Methodology: Processes, Models and Diagrams. In: 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems. ACM Press, New York (2002)
19. Henderson-Sellers, B., Giorgini, P.: Agent-Oriented Methodologies. Idea Group Publishing, USA (2005)
20. Holvoet, T., Valckenaers, P.: Exploiting the Environment for Coordinating Agent Intentions. In: E4MAS, Hakodate, Japan. LNCS, vol. 4389, pp. 51–66. Springer, Heidelberg (2006)
21. Jazayeri, M., Ran, A., van der Linden, F.: Software Architecture for Product Families. Addison Wesley Longman Inc. Redwood City,CA, USA (2000)
22. Kendall, E., Jiang, C.: Multiagent System Design Based on Object Oriented Patterns. Journal of Object Oriented Programming 10(3), 41–47 (1997)
23. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241. Springer, Heidelberg (1997)
24. Kolp, M., Giorgini, P., Mylopoulos, J.: A Goal-Based Organizational Perspective on Multi-agent Architectures. In: Meyer, J.-J.C., Tambe, M. (eds.) ATAL 2001. LNCS (LNAI), vol. 2333, pp. 128–140. Springer, Heidelberg (2002)
25. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design. Prentice-Hall, Englewood Cliffs (2002)
26. McConell, S.: Rapid Development: Taming Wild Software Schedules. Microsoft Press, Redmond, Washington (1996)
27. Olumofin, F., Misic, V.: Extending the ATAM Architecture Evaluation to Product Line Architectures. In: 5th IEEE-IFIP Conference on Software Architecture, Pittsburgh, Pennsylvania, USA (2005)
28. Rao, A., Georgeff, M.: BDI Agents: From Theory to Practice. In: 1st International Conference on Multiagent Systems, San Francisco, California, USA. The MIT Press, Cambridge (1995)
29. Schelfthout, K., Holvoet, T.: Views: Customizable abstractions for context-aware applications in MANETs. In: Software Engineering for Large-Scale Multi-Agent Systems, St. Louis, USA (2005)
30. Schelfthout, K., Weyns, D., Holvoet, T.: Middleware that Enables Protocol-Based Coordination Applied in Automatic Guided Vehicle Control. IEEE Distributed Systems Online 7(8) (2006)
31. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs (1996)

32. O. Shehory. Architectural Properties of MultiAgent Systems. Technical Report CMU-RI-TR-98-28, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1998.
33. Steegmans, E., Weyns, D., Holvoet, T., Berbers, Y.: A Design Process for Adaptive Behavior of Situated Agents. In: Odell, J.J., Giorgini, P., Müller, J.P. (eds.) AOSE 2004. LNCS, vol. 3382. Springer, Heidelberg (2005)
34. Weyns, D.: An Architecture-Centric Approach for Software Engineering with Situated Multiagent Systems. Ph.D, Katholieke Universiteit Leuven (2006)
35. Weyns, D., Boucké, N., Holvoet, T.: Gradient Field Based Transport Assignment in AGV Systems. In: 5th International Joint Conference on Autonomous Agents and Multi-Agent Systems, AAMAS, Hakodate, Japan (2006)
36. Weyns, D., Helleboogh, A., Holvoet, T.: The Packet-World: a Test Bed for Investigating Situated Multi-Agent Systems. In: Agent-based applications, platforms, and development kits. Whitestein Series in Software Agent Technology (2005)
37. Weyns, D., Holvoet, T.: Model for Simultaneous Actions in Situated Multiagent Systems. In: Schillo, M., Klusch, M., Müller, J., Tianfield, H. (eds.) Multiagent System Technologies. LNCS (LNAI), vol. 2831. Springer, Heidelberg (2003)
38. Weyns, D., Holvoet, T.: Formal Model for Situated Multi-Agent Systems. Fundamenta Informaticae 63(1-2), 125–158 (2004)
39. Weyns, D., Holvoet, T.: Regional Synchronization for Situated Multi-agent Systems. In: Mařík, V., Müller, J.P., Pěchouček, M. (eds.) CEEMAS 2003. LNCS (LNAI), vol. 2691. Springer, Heidelberg (2003)
40. Weyns, D., Holvoet, T.: Architectural Design of an Industrial AGV Transportation System with a Multiagent System Approach. In: Software Architecture Technology User Network Workshop, SATURN, Pittsburg, USA, Software Engineering Institute, Carnegie Mellon University (2006)
41. Weyns, D., Holvoet, T.: From Reactive Robotics to Situated Multiagent Systems: A Historical Perspective on the Role of Environment in Multiagent Systems. In: Dikenelli, O., Gleizes, M.-P., Ricci, A. (eds.) ESAW 2005. LNCS (LNAI), vol. 3963. Springer, Heidelberg (2006)
42. Weyns, D., Omicini, A., Odell, J.: Environment as a First-Class Abstraction in Multiagent Systems. Autonomous Agents and Multi-Agent Systems 14(1) (2007)
43. Weyns, D., Schelfthout, K., Holvoet, T., Lefever, T.: Decentralized control of E'GV transportation systems. In: 4th Joint Conference on Autonomous Agents and Multiagent Systems, Industry Track, Utrecht, The Netherlands. ACM Press, New York (2005)
44. Weyns, D., Steegmans, E., Holvoet, T.: Integrating Free-Flow Architectures with Role Models Based on Statecharts. In: Choren, R., Garcia, A., Lucena, C., Romanovsky, A. (eds.) Software Engineering for Multi-Agent Systems III. LNCS, vol. 3390. Springer, Heidelberg (2005)
45. Weyns, D., Steegmans, E., Holvoet, T.: Protocol Based Communication for Situated Multi-Agent Systems. In: 3th Joint Conference on Autonomous Agents and Multi-Agent Systems, New York, USA. IEEE Computer Society Press, Los Alamitos (2004)
46. Weyns, D., Steegmans, E., Holvoet, T.: Towards Active Perception in Situated Multi-Agent Systems. Applied Artificial Intelligence 18(9-10), 867–883 (2004)
47. Wyns, J., Van Brussel, H., Valckenaers, P., Bongaerts, L.: Workstation Architecture in Holonic Manufacturing Systems. In: 28th CIRP International Seminar on Manufacturing Systems, Johannesburg, South Africa (1996)
48. Yu, E.: Modelling Strategic Relationships for Process Reengineering, PhD Dissertation: University of Toronto, Canada (1995)