A Reference Architecture for 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 reference architecture integrates a set of architectural patterns that have proven their value for a family of applications. Such family of applications is characterized by specific functionality and quality requirements. A reference architecture provides a blueprint for developing software architectures for applications that share that common base. As such, a reference architecture provides a means for large-scale reuse of architectural design.

This paper gives an overview of a reference architecture for situated multiagent systems we have developed in our research. We discuss various architectural views of the reference architecture. Per view, we zoom in on the main view packets, each of them containing a bundle of information of a part of the reference architecture. For each view packet we explain the rationale for the design choices that were made and we give built-in mechanisms that describe how the view packet can be exercised to build a concrete software architecture. We illustrate the use of the reference architecture with an excerpt of the software architecture of an industrial AGV transportation system.

1 Introduction

A reference architecture embodies a set of architectural best practices gathered from the design and development of a family of applications with similar characteristics and system requirements [28,6]. A reference architecture provides an asset base architects can draw from when developing software architectures for new systems that share the common base of the reference architecture. Applying the reference architecture to develop new software architectures will yield valuable input that can be used to update and refine the reference architecture. As such, a reference architecture provides a means for large-scale reuse of architectural design.

In this paper, we give an overview of the reference architecture for situated multiagent systems we have developed in our research. We start with an introductory section that explains the reference architecture rationale and sketches the background of the architecture. Next, we present the reference architecture. The architecture documentation consists of three views that describe the reference architecture from different perspectives. To illustrate the use of the reference

D. Weyns, H.V.D. Parunak, and F. Michel (Eds.): E4MAS 2006, LNAI 4389, pp. 1–40, 2007.

architecture, we give an excerpt of the software architecture of an industrial AGV transportation system in which we have used the reference architecture for architectural design. The paper concludes with an overview of related work and conclusions.

2 Rationale and Background

In this section, we explain the reference architecture rationale. We summarize the main characteristics and requirements of the target application domain of the reference architecture and give a brief overview of the development process of the architecture. Finally, we explain how the reference architecture documentation is organized.

2.1 Reference Architecture Rationale

The general goal of the reference architecture is to support the architectural design of situated multiagent systems. Concrete motivations are:

- *Integration of mechanisms.* In our research, we have developed several advanced mechanisms of adaptivity for situated agents, including selective perception [48], advanced behavior-based action-selection mechanisms with roles and situated commitments [46,44], and protocol-based communication [47]. To build a concrete application these mechanisms have to work together. The reference architecture integrates the different mechanisms. 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.
- *Blueprint for architectural design.* The reference architecture generalizes common functions and structures from various experimental applications we have studied and built. This generalized architecture provides a reusable design artifact, it facilitates deriving new software architectures for systems that share the common base more reliably and cost effectively. On the one hand, the reference architecture defines constraints that incarnate the common base. On the other hand, the architecture defines variation mechanisms that provide the necessary variability to instantiate software architectures for new systems.
- *Reification of knowledge and expertise.* 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 multiagent systems we have developed.

2.2 Characteristics and Requirements of the Target Application Domain of the Reference Architecture

The reference architecture for situated multiagent systems supports the architectural design of a family 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.

2.3 Development Process of the Reference Architecture

The reference architecture for situated multiagent systems is the result of an iterative research process of exploration and validation. During our research, we have studied and built various experimental applications that share the above specified characteristics in different degrees. We extensively used the Packet–World as a study case for investigation and experimentation. [40,42] investigate agents' actions in the Packet–World. [39] studies various forms of stigmergic coordination. [44] focuses on the adaptation of agent behavior over time. [32] yields valuable insights on the modelling of state of agents and the environment, selective perception, and protocol–based communication. Another application we have used in our research is a prototypical peer-to-peer file sharing system [48,20]. This application applies a pheromone–based approach for the coordination of agents that move around in a dynamic network searching for files. [38,8,31] study a field-based approach for task assignment to automatic guided vehicles that have to operate in a dynamic warehouse environment. Finally, [46] studies several experimental robotic applications. The particular focus of these robotic applications is on the integration of roles and situated commitments in behavior–based action selection mechanisms.

Besides these experimental applications, the development of the reference architecture is considerably based on experiences with an industrial logistic transportation system for warehouses [45,43,9].

In the course of building the various applications, we derived common functions and structures that provided architectural building blocks for the reference architecture. The reference architecture integrates the different agent and environment functionalities and maps these functionalities onto software elements and relationships between the elements. The software elements make up a system decomposition that cooperatively implement the functionalities. This system

decomposition—the reference architecture—provides a blueprint for instantiating target systems that share the common base of the reference architecture.

2.4 Organization of the Reference Architecture Documentation

The architecture documentation describes the various architectural views of the reference architecture [13]. The documentation includes a module decomposition view and two component and connector views: shared data and communicating processes. Each view is organized as a set of view packets. A view packet is a small, relatively self-contained bundle of information of the reference architecture, or a part of the architecture. The documentation of a view starts with a brief explanation of the goal of the view and a general description of the view elements and relationships between the elements. Then the view packets of the view are presented. Each view packet consists of a primary presentation and additional supporting information. The primary presentation shows the elements and their relationships in the view packet. For the module decomposition view, the primary presentations are textual in the form of tables. The primary presentations of other views are graphical with a legend that explain the meaning of the symbols.

The supporting information explains the architectural elements in the view packet. Each view packet gives a description of the architectural elements with their specific properties. In addition to the explanation of the architectural elements, the supporting information describes variation mechanisms for the view packet and explains the architecture rationale of the view packet. Variation mechanisms describe how the view packet can be applied to build a software architecture for a concrete system. The architecture rationale explains the motivation for the design choices that were made for the view packet.

The documentation of the reference architecture presented in this paper is descriptive. Concepts and mechanisms are introduced briefly and illustrated with examples. The interested reader finds elaborated explanations in the added references. For a detailed formal specification of the various architectural elements, we refer to [37].

3 Module Decomposition View

The module decomposition view shows how the situated multiagent system is decomposed into manageable software units. The elements of the module decomposition view are *modules*. A module is an implementation unit of software that provides a coherent unit of functionality. The relationship between the modules is *is–part–of* that defines a part/whole relationship between a submodule and the aggregate module. Modules are recursively refined conveying more details in each decomposition step.

The basic criteria for module decomposition is the achievement of quality attributes. For example, changeable parts of a system are encapsulated in separate modules, supporting modifiability. Another example is the separation of

functionality of a system that has higher performance requirements from other functionality. Such a decomposition allows to apply different tactics to achieve the required performance throughout the various parts of the system. However, other criteria can be drivers for a decomposition of modules as well. For example, in the reference architecture, a distinction is made between common modules that are used in all systems derived from the reference architecture, and variable modules that differ across systems. This decomposition results in a clear organization of the architecture, supporting efficient design and implementation of systems with the reference architecture.

Modules in the module decomposition view include a description of the interfaces of the modules that document how the modules are used in combination with other modules. The interface descriptions distinguish between provided and required interfaces. A provided interface specifies what functionality a module offers to other modules. A required interface specifies what functionality a module needs from other modules; it defines constrains of a module in terms of the services a module requires to provide its functionality.

The reference architecture provides three view packets of the module decomposition view. We start with the top-level decomposition of the situated multiagent system. Next, we show the primary decomposition of an agent. We conclude with the primary decomposition of the application environment.

3.1 Module Decomposition View Packet 1: Situated Multiagent System

Primary Presentation

System	**Subsystem**
Situated Multiagent System	Agent
	Application Environment

Elements and their Properties. A Situated Multiagent System is decomposed in two subsystems: Agent and Application Environment. We explain the functionalities of both modules in turn.

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 multiagent system 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 multiagent system 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. 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.

Interface Descriptions. Figure 1 gives an overview of the interfaces of the agent subsystem and the application environment subsystem.

The `Sense` interface enables an agent to sense the environment resulting in a representation, `Send` enables an agent to send messages to other agents, and `Influence` enables an agent to invoke influences in the environment. Influences are attempts of agents to modify the state of affairs in the environment. These interfaces are provided by the application environment.

The application environment requires the interface `Receive` to deliver messages to agents. Furthermore, the application environment requires the interface

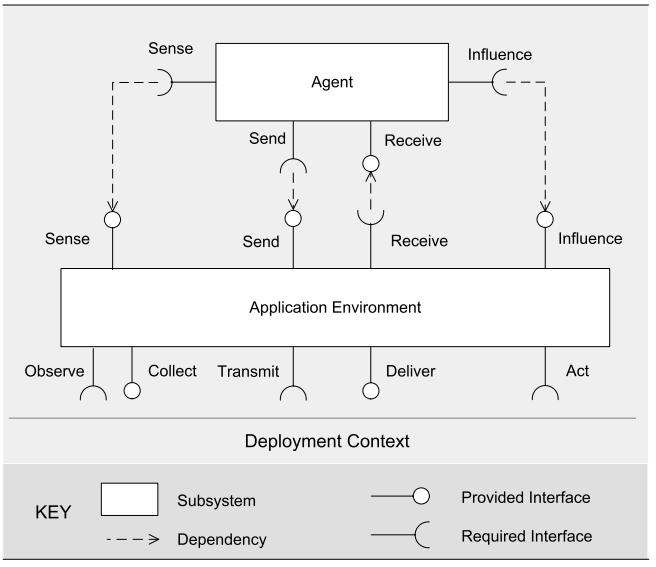

Fig. 1. Interfaces of agent, application environment, and deployment context

`Observe` from the deployment context to observe particular resources, `Transmit` to send messages to agents located on other nodes, and `Act` to modify the state of external resources (based on influences invoked by agents).

Finally, the deployment context requires the interface `Collect` to enable the collection of state from the application environment (requested by application environment instances in a distributed setting), and the interface `Deliver` to deliver the incoming messages to the agents.

Variation Mechanisms. There are four variation mechanisms for this view packet:

M1 *Definition of Agent Types.* Depending on the specific application requirements different agent types may be required. Agent types are characterized by the capabilities of the agents reflected in different internal structures. Variations on agent types are discussed in subsequent view packets and views, see sections 3.2, 4.1, and 5.

M2 *Definition of Application Environment Types.* In a distributed setting, different application environment types may be required that are deployed on different nodes. Application environment types differ in the functionality they provide to the agents reflected in different internal structures. Variations in application environment types are discussed in subsequent view packets and views, see sections 3.3, 4.2, and 5.

M3 *Definition of the Domain Ontology.* The ontology defines the terminology for the application domain. Defining an ontology includes the specification of the various domain concepts and the relationships between the concepts.

The domain ontology serves as a basis for the definition of the knowledge of the agents and the state of the application environment, see the variation mechanisms SD1 and SD2 of the component and connector shared data view in section 4.

M4 *Definition of the Interaction Primitives of the Deployment Context.* To enable the multiagent system software deployed on a node to interact with the deployment context, the various interaction primitives with the deployment context have to be concretized according to the application at hand. We distinguish between three types of interaction primitives.

(1) Observation primitives enable the multiagent system software deployed on a node to observe external resources and collect data from other nodes. An observation primitive indicates which resource is observed and what type of information should be observed.
(2) Action primitives enable to access external resources. An action primitive indicates the target resource and the type of action.
(3) Communication primitives enable to transmit low-level formatted messages via the deployment context. A low-level formatted message is a data structure that represents a message exchanged between a sender and one or more addressees and that is transmitted via the deployment context.

Design Rationale. The main principles that underly the decomposition of a situated multiagent system are:

- *Decentralized control.* In a situated multiagent system, control is divided among the agents situated in the application environment. Decentralized control is essential to cope with the inherent locality of activity, which is a characteristic of the target applications of the reference architecture, see section 2.2.
- *Self-management.* In a situated multiagent system self-management is essentially based on the ability of agents to adapt their behavior. Self-management enables a system to manage the dynamic and changing operating conditions autonomously, which is an important requirement of the target applications of the reference architecture, see section 2.2.

However, the decentralized architecture of a situated multiagent system implies a number of tradeoffs and limitations.

- Decentralized control typically requires more communication. The performance of the system may be affected by the communication links between agents.
- There is a trade-off between the performance of the system and its flexibility to handle disturbances. A system that is designed to cope with many disturbances generally needs redundancy, usually to the detriment of performance, and vice versa.
- Agents' decision making is based on local information only, which may lead to suboptimal system behavior.

These tradeoffs and limitations should be kept in mind throughout the design and development of a situated multiagent system. Special attention should be payed to communication which could impose a major bottleneck.

Concerns not Covered. We touch on a number of other concerns that are not covered by the reference architecture.

Crosscutting Concerns. Concerns such as security, monitoring, and logging usually crosscut several architecture modules. Crosscutting concerns in multiagent systems are hardly explored and are open research problems. An example of early research in this direction is [17]. That work applies an aspect-oriented software engineering approach, aiming to integrate crosscutting concerns in an application in a non-invasive manner. As most current research on aspect-oriented software development, the approach of [17] is mainly directed at the identification and specification of aspects at the programming level. Recently, the relationship between aspects and software architecture became subject of active research, see e.g. [4,35,14].

Human-Software Interaction. The reference architecture does not explicitly handle human-software interaction. Depending on the application domain, the role of humans in multiagent systems can be very diverse. In some applications humans can play the role of agents and interact directly—or via an intermediate wrapper—with the application environment. In other applications, humans can be part of the deployment context with which the multiagent system application interacts.

3.2 Module Decomposition View Packet 2: Agent

Primary Presentation

Subsystem	**Module**
Agent	Perception
	Decision Making
	Communication

Elements of the View. The Agent subsystem is decomposed in three modules: Perception, Decision Making and Communication. We discuss the responsibilities of each module in turn.

Perception is responsible for collecting runtime information from the environment. The perception module supports selective perception [48]. Selective perception enables an agent to direct its perception according to its current tasks. To direct its perception agents select 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. While a focus enables an agent to observe the environment for a particular type of information, a filter enables the agent to direct its attention within the sensed information.

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 [41].

To enable situated agents to set up collaborations, behavior-based action selection mechanisms are extended with the notions of *role* and *situated commitment* [46,33,32,47]. 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 [47]. 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 [5] 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. The ontology used for communication is typically a part of the integral ontology of the application domain, see section 3.1.

Interface Descriptions. The interface descriptions specify how the modules of an agent are used with one another, see Fig. 2. The interfacing with the data repositories is discussed in section 4.1.

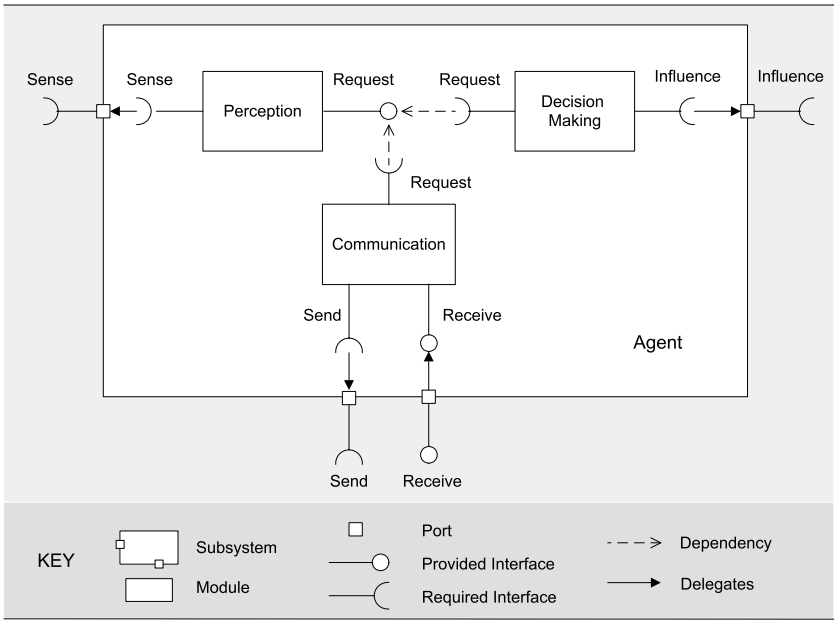

Fig. 2. Interfaces of the agent modules

The provided `Request` interface of the perception module enables decision making and communication to request a perception of the environment. To sense the environment according to their current needs, decision making and communication pass on a focus and filter selector to the perception module. Such a selector specifies a set of foci and filters that the perception module uses to sense the environment selectively [48].

The provided interface of agent, `Receive`, delegates for processing incoming messages to the provided `Receive` interface of the communication module. The ports decouple the internals of the agent subsystem from external elements.

The perception module's required `Sense` interface is delegated to the agent's required `Sense` interface. Sensing results in a representation of the environment according to the selected foci. Similarly, the `Send` interface of the communication module and the `Influence` interface of the decision making module are delegated to the required interfaces of agent with the same name.

Variation Mechanisms. This view packet provides the following variation mechanisms:

M5 *Omission of the Communication module.* For agents that do not communicate via message exchange, the communication module can be omitted. An example is an ant-like agent system in which the agents communicate via the manipulation of marks in the environment.

M6 *Definition of Foci and Focus Selectors.* Foci enable agents to sense the environment selectively. The definition of the foci in the agent system includes the specification of the kind of data each focus targets, together with the scoping properties of each focus. The definition of focus selectors includes the specification of the various combinations of foci that can be used to sense the environment.

M7 *Definition of Representations.* Sensing the environment results in representations. Representations are defined by means of data structures that represent elements and resources in the environment. The definition of representations must comply to the ontology defined for the domain, see variation mechanism M3 in section 3.1.

M8 *Definition of Filters and Filter Selectors.* Filters can be used by agents to filter perceived data. The definition of the filters in the agent system includes the specification of the kind of data each filter aims to filter and the specific properties of each filter. The definition of filter selectors includes the specification of the various combinations of filters that can be used to filter percepts.

M9 *Definition of Influences.* Influences enable agents to modify the state of affairs in the environment. The definition of an influence includes the specification of an operation that is provided by the application environment and that can be invoked by the agents.

M10 *Definition of Roles and Situated Commitments and Specification of an Action Selection Mechanism.* Each role in the agent system is defined by a unique name and a description of the semantics of the role in terms of the influences that can be selected in that role as well as the relationship of the role to other roles in the agent system. Each situated commitment in the agent system is defined by a unique name and a description of the semantics of the commitment in terms of roles defined in the agent system. Situated agents use a behavior-based action selection mechanism. Depending on the system requirements a particular action selection mechanism has to be defined. Roles and situated commitments have to be mapped onto the chosen action selection mechanism. [32] discusses an example where roles and situated commitments are mapped onto a free-flow decision making tree.

M11 *Definition of the Communication Language and the Ontology.* The communication language defines the format of messages. The definition of the communication language includes the specification of identities for agents and conversations, the specification of the various performatives of the language, and the format of the content of messages. The definition of the ontology for communication includes the specification of the vocabulary of words that represent the domain concepts used in messages and

the relationships between the concepts. The ontology for communication is typically a part of the integral domain ontology, see variation mechanism M3 in section 3.1.

M12 *Definition of Communication Protocols.* The definition of a concrete communication protocol includes the specification of various steps of the protocol, i.e. the conditions and the effects for each step in the protocol [47]. An important aspect of this latter is the activation/deactivation of situated commitments. Statecharts [18,3] are one possible approach to specify a communication protocol.

Design Rationale. Each module in the decomposition encapsulates a particular functionality of the agent. By minimizing the overlap of functionality among modules, the architect can focus on one particular aspect of the agent's functionality. Allocating different functionalities of an agent to separate modules results in a clear design. It helps to accommodate change and to update one module without affecting the others, and it supports reusability.

Perception on Command. Selective perception enables an agent to focus its attention to the relevant aspects in the environment according to its current tasks. 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.

Coordination between Decision Making and Communication. The overall behavior of the agent is the result of the coordination of two modules: 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. Decision making and communication coordinate to complete the agent's tasks. 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.

Ensuring that both decision making and communication behave in a coordinated way requires a careful design. On the other hand, 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 of separating communication from performing actions 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, but it applies to any application in which the time

required for performing actions in the environment differs significantly from the time to communicate messages. Separation of communication from performing actions enables agents to reconsider the coordination of their behavior while they perform actions, improving adaptability and efficiency.

3.3 Module Decomposition View Packet 3: Application Environment

Primary Presentation

Subsystem	**Module**
Application Environment	Representation Generator
	Observation & Data Processing
	Interaction
	Low-Level Control
	Communication Mediation
	Communication Service
	Synchronization & Data Processing

Elements and their Properties. The Application Environment subsystem is decomposed in seven modules. We discuss the responsibilities of each of the modules in turn.

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. A perception law defines restrictions on what an agent can sense from the environment with a set of foci.

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. Rather than delivering raw data retrieved from the observation, 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. Action laws put restrictions on the influences invoked by the agents, representing domain specific constraints on agents' actions. 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. 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, a FIPA ACL message [16] consists of a header with the message performative (inform, request, propose, etc.), followed by the subject of this performative, i.e. the content of the message that is described in a content language that is based on a shared ontology. Such message descriptions 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]. The synchronization & data processing module 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.

Interface Descriptions. The interface descriptions specify how the modules of the application environment are used with one another, see Fig. 3. The interfacing with data repositories of the application environment is discussed in section 4.2.

The `Sense` interface of the application environment delegates perception requests to the `Sense` interface of the perception generator. To observe resources in the deployment context, the perception generator's required interface `CollectData` depends on the `CollectData` interface that is provided by the observation & data processing module. The required interface `Observe` of observation & data processing is delegated to `Observe` interface of the application environment. The provided interface `Collect` of the application environment delegates requests for state of the application environment to the `Collect` interface of the observation & data processing module. The data that results from the observation of resources in the deployment context and possible state collected from other nodes is processed by the observation & data processing module and passed to the perception generator that generates a representation for the requesting agent.

For its functioning, the synchronization & data processing module requires the interface `Observe`. The processing of this interface is delegated to the `Observe` interface of the application environment. Synchronization & data processing provides the `Collect` interface to allow sharing of data among nodes in a distributed setting. This interface depends on the `Collect` interface provided by the application environment.

The `Send` interface of the application environment enables agents to send messages to other agents. The application environment delegates this interface to the

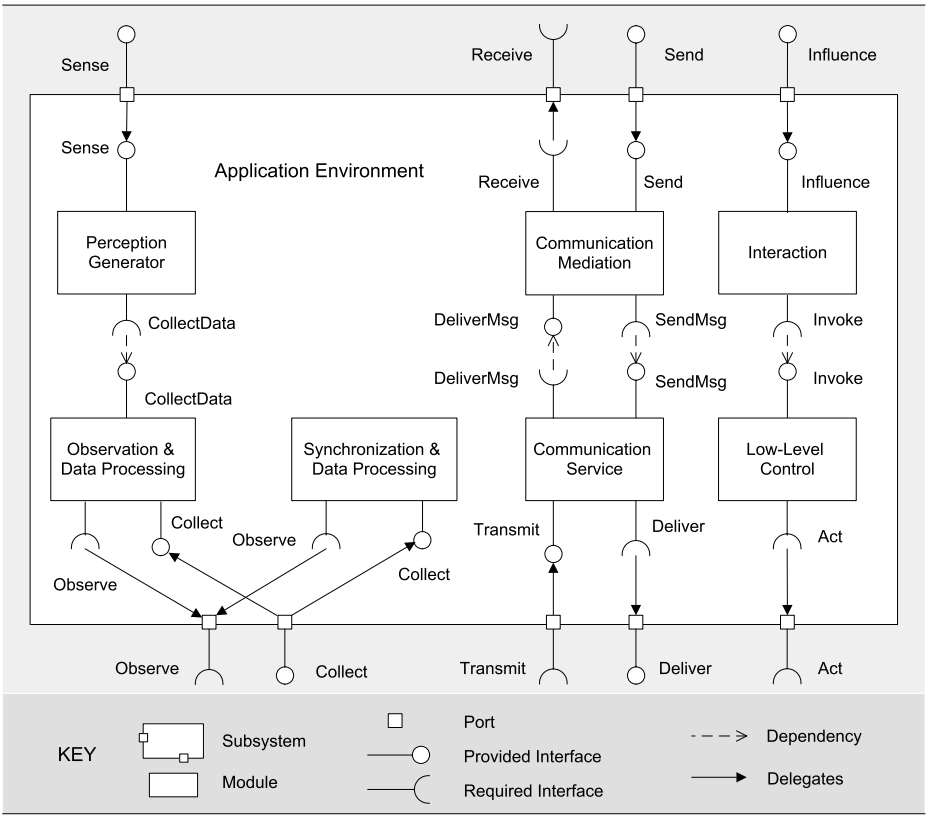

Fig. 3. Interfaces of the application environment modules

`Send` interface of communication mediation that mediates the communicative interaction. Communication mediation depends on the provided interface `SendMSg` of the communication service to convert messages into a low-level format and transmit them via the deployment context. For this latter, the communication service delegates to the `Deliver` interface of the application environment that depends on the message transfer infrastructure of the deployment context. The `Transmit` interface of the communication service delegates the transmission of messages to the `Transmit` interface of the application environment. The application environment provides the `Deliver` interface to deliver incoming messages. The `Deliver` interface of the application environment delegates incoming messages to the `Deliver` interface of the communication service. This latter converts the messages into an appropriate format for agents and depends on the `DeliverMsg` interface of communication mediation to deliver the messages. The `Receive` interface of communication mediation delegates the delivering of messages to the `Receive` interface of the application environment that passes on the messages to the addressees. The provided interface `Influence` of the

application environment enables agents to invoke influences in the environment. For influences that attempt to modify the state of resources in the deployment context, the interaction module's required interface `Invoke` depends on the interface `Invoke` provided by the low-level control module. This latter interface provides the functionality to convert influences into low-level action primitives of the deployment context. The `Act` interface of the low-level control module delegates the actions to external resources to the `Act` interface of the application environment that invokes the actions in the deployment context.

Variation Mechanisms. This view packet provides the following variation mechanisms:

M13 *Omission of Observation, Synchronization, and Low-Level Control.* For applications that do not interact with external resources, the observation, synchronization, and low-level control modules can be omitted. For such applications, the environment is entirely virtual.

M14 *Omission of Communication Mediation and Communication Service.* For agent systems in which agents do not communicate via message exchange, the modules related to message exchange can be omitted, see also variation mechanism M5 in section 3.2.

M15 *Omission of Synchronization & Data Processing.* For multiagent system applications where no synchronization of state between the application environment and the deployment context and/or between nodes is required, the synchronization & data processing module can be omitted.

M16 *Definition of Observations.* Observations enable the multiagent system to collect data from the deployment context. The definition of an observation includes the specification of the kind of data to be observed in the deployment context together with additional properties of the observation.

The definition of the laws for perception, interaction, and communication is discussed in the shared data view, see section 4.2.

Design Rationale. The decomposition of the application environment can be considered in two dimensions: horizontally, i.e. a decomposition based on the distinct ways agents can access the environment; and vertically, i.e. a decomposition based on the distinction between the high-level interactions between agents and the application environment, and the low-level interactions between the application environment and the deployment context. The decomposition is schematically shown in Fig. 4.

The horizontal decomposition of the application environment consists of three columns that basically correspond to the various ways agents can access the environment: perception, communication, and action. An agent can *sense* the environment to obtain a *representation* of its vicinity, it can exchange *messages* with other agents, and an agent can invoke an *influence* in the environment attempting to modify the state of affairs in the environment.

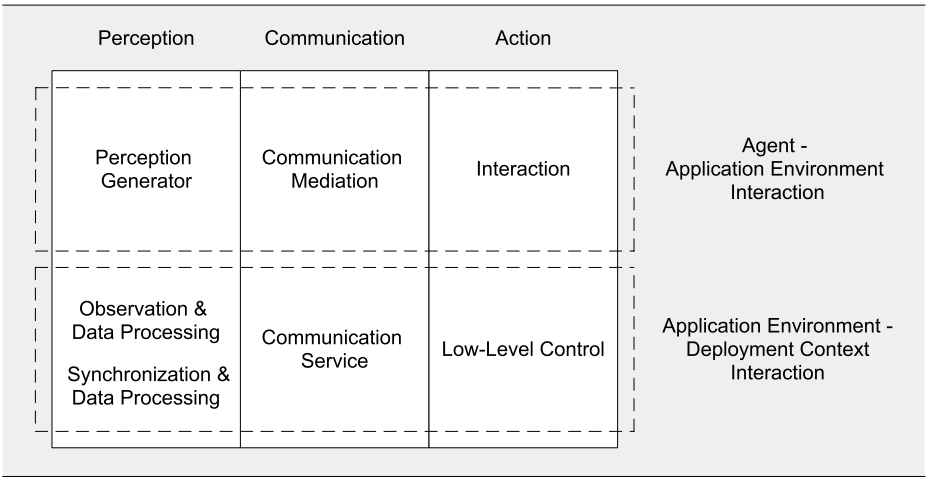

Fig. 4. Decomposition application environment

The vertical decomposition of the application environment consists of two rows. The top row deals with the access of agents to the application environment and includes representation generator, communication mediation, and interaction. The specification of activities and concepts in the top row is the same as those used by the agents. The top row defines the various laws that constrain the activity of agents in the environment. The bottom row deals with the interaction of the application environment with the deployment context and consists of observation and synchronization with data processing, communication service, and low-level control. The functionality related to the low-level interactions of the application environment includes: (1) support for the conversion of high-level activity related to agents into low-level interactions related to the deployment context and vice versa, and (2) support for pre-processing of resource data to transfer the data into a higher-level representation useful to agents, (3) interaction and synchronization among different nodes in a distributed setting.

The two-dimensional decomposition of the application environment yields a flexible modularization that can be tailored to a broad family of application domains. For instance, for applications that do not interact with an external deployment context, the bottom layer of the vertical decomposition can be omitted. For applications in which agents interact via marks in the environment but do not communicate via message exchange, the column in the horizontal decomposition that corresponds to message transfer (communication and communication service) can be omitted.

Each module of the application environment is located in a particular column and row and is assigned a particular functionality. Minimizing the overlap of functionality among modules, helps the architect to focus on one particular

aspect of the functionality of the application environment. It supports reuse, and it further helps to accommodate change and to update one module without affecting the others.

4 Component and Connector Shared Data View

The shared data view shows how the situated multiagent system is structured as a set of data accessors that read and write data in various shared data repositories. The elements of the shared data view are *data accessors*, *repositories*, and the *connectors* between the two. Data accessors are runtime components that perform calculations that require data from one or more data repositories. Data repositories mediate the interactions among data accessors. A shared data repository can provide a trigger mechanism to signal data consumers of the arrival of interesting data. Besides reading and writing data, a repository may provide additional support, such as support for concurrency and persistency. The relationship of the shared data view is *attachment* that determines which data accessors are connected to which data repositories [13]. Data accessors are attached to connectors that are attached to a data store.

The reference architecture provides two view packets of the shared data view. First, we zoom in on the shared data view packet of agent, then we discuss the view packet of the application environment. The data accessors in this view are runtime instances of modules we have introduced in the module decomposition view. We use the same names for the runtime components and the modules (components' names are proceeded by a colon).

4.1 C & C Shared Data View Packet 1: Agent

Primary Presentation. The primary presentation is shown in Fig. 5.

Elements and their Properties. The data accessors of the Agent view packet are Perception, Decision Making and Communication. These data accessors are runtime instances of the corresponding modules described in section 3.2. The data accessors share the Current Knowledge repository.

The **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. The communication and decision making components can read and write data from the repository. The perception component maintains the agent's knowledge of the surrounding environment. To update the agent's knowledge of the environment, both the communication and decision making components can trigger the perception component to sense the environment, see the module view of agent in section 3.2.

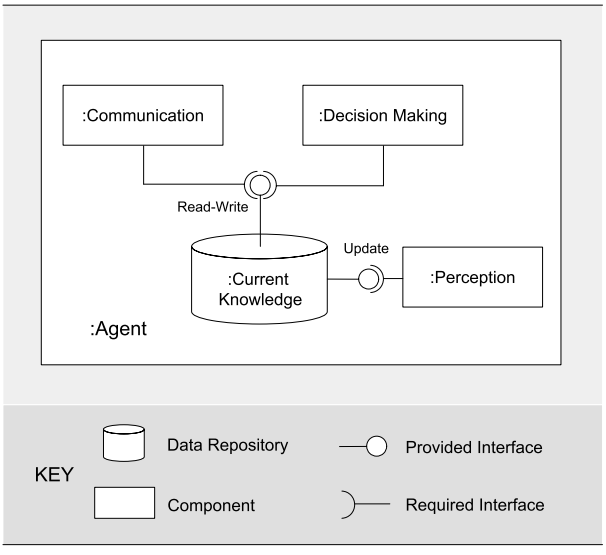

Fig. 5. Shared data view of an agent

Interface Descriptions. Fig. 5 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. Unless stated otherwise, we assume that the 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.

Variation Mechanisms. This view packet provides four variation mechanisms:

SD1 *Definition of Current Knowledge.* Definition of current knowledge includes the definition of the state of the agent and the specification of the knowledge repository. The definition of the state of the agent has to comply to the ontology that is defined for the multiagent system application, see variation mechanism M3 in section 3.1. The specification of the knowledge repository includes various aspects such as the specification of a policy for concurrency, specification of possible event mechanisms to signal data consumers, support for persistency of data, and support for transactions. The concrete interpretation of these aspects depends on the specific requirements of the application at hand.

Design Rationale. The shared data style decouples the various components of an agent. Low coupling improves modifiability (changes in one element do not affect other elements or the changes have only a local effect) and reuse (elements are not dependent on too many other elements). Low coupled elements usually have clear and separate responsibilities, which makes the elements better to understand in isolation. Decoupled elements do not require detailed knowledge about the internal structures and operations of the other elements. Due to the concurrent access of the repository, the shared data style requires special efforts to synchronize data access.

Both communication and decision making delegate perception requests to the perception component. The perception component updates the agent knowledge with the information derived from perceiving the environment. The current knowledge repository makes the up-to-date information available for the communication and decision making component. By sharing the knowledge, both components can use the most actual data to make decisions.

The current knowledge repository enables the communication and decision making components to share data and to communicate indirectly. This approach allows both components to act in parallel and at a different pace, improving efficiency and adaptability (see also the design rationale of the module decomposition view of agent in section 3.2).

An alternative for the shared data style is a design where each component encapsulates its own state and provides interfaces through which other elements get access to particular information. However, since a lot of state is shared between the components of an agent (examples are the state that is derived from perceiving the environment and the state of situated commitments), such a design would increase dependencies among the components or imply the duplication of state in different components. Furthermore, such duplicated state must be kept synchronized among the components.

4.2 C & C Shared Data View Packet 2: Application Environment

Primary Presentation. The primary presentation is depicted in Fig. 6.

Elements and their Properties. The Application Environment consists of various data accessors that are attached to two repositories: State and Laws. The data accessors are runtime instances of the corresponding modules introduced in section 3.3.

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

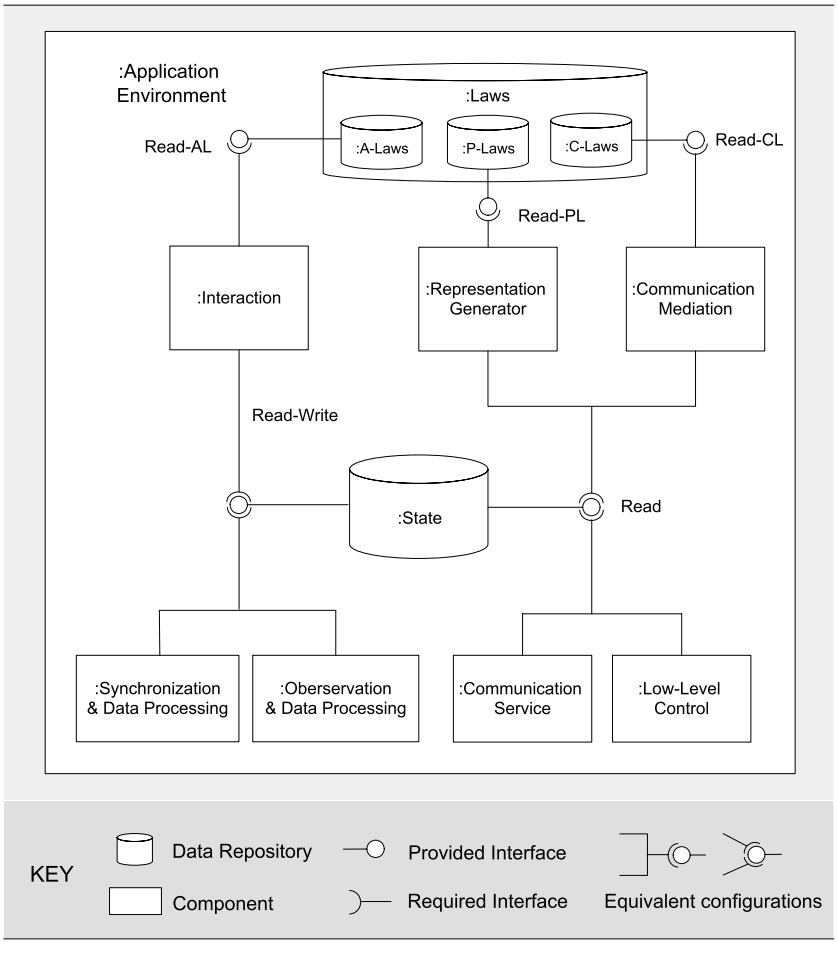

Fig. 6. Shared data view of the application environment

include agent-specific data, such as the agents' identities, the positions of the agents, and tags used for coordination purposes.

To perform their functionalities, interaction, synchronization & data processing, and observation & data processing can read and write state of the application environment. Representation generator, communication mediation and communication service, and low-level control only need to read state of the state repository to perform their functionalities.

The **Laws** repository contains the various laws that are defined for the application at hand. The laws repository is divided in three sub-repositories, one with the perception laws, one with the action laws, and one with communication laws. Each of these sub-repositories is attached to the component responsible for the corresponding functionality.

Interface Descriptions. Fig. 6 shows the interconnections between the state repositories and the internal components of the application environment.

The state repository exposes two interfaces. The provided interface `Read` enables attached components to read state of the repository. The `Read-Write` interface enables the attached components to access and modify the application environment's state.

The laws repository exposes three interfaces to read the various types of laws: `Read-AL`, `Read-PL`, and `Read-CL`. These provided interfaces enable the attached components to consult the respective types of laws.

Variation Mechanisms. This view packet provides one variation mechanism:

SD2 *Definition of State.* The definition of state includes the definition of the actual state of the application environment and the specification of the state repository. The state definition has to comply to the ontology that is defined for the application domain, see variation mechanism M3 in section 3.1. The specification of the state repository includes various aspects such as the specification of a policy for concurrency, specification of possible event mechanisms to signal data consumers, support for persistency of data, and support for transactions. As for the definition of the current knowledge repository of an agent, the concrete interpretation of these aspects depends on the specific requirements of the application domain at hand.

SD3 *Definition of Action Laws.* Action laws impose application specific constraints on agents' influences in the environment. An action law defines restrictions on what kinds of manipulations agents can perform in the environment for a particular influence. The constraints imposed by an action law can be defined relative to the actual state of the environment. For example, when an agent injects a tuple in network, the distribution of the tuple can be restricted based on the actual cost for the tuple to propagate along the various links of the network.

SD4 *Definition of Perception Laws.* Perception laws impose application specific constraints on agents' perception of the environment. Every perception law defines restrictions on what can be sensed from the current state of the environment for a particular focus. The constraints imposed by a perception law can be defined relative to the actual state of the environment. For example, restrictions on the observation of local nodes in a mobile network can be defined as a function of the actual distance to the nodes in the network.

SD5 *Definition of Communication Laws.* Communication laws impose application specific constraints on agents' communicative interactions in the environment. A communication law defines restrictions on the delivering of messages. The constraints imposed by a communication law can be defined relative to the actual state of the environment. For example, the delivering of a broadcast message in a network can be restricted to addressees that are located within a particular physical area around the sender.

Design Rationale. The motivations for applying the shared data style in the design of the application environment are similar as for the design of an agent. The shared data style results in low coupling between the various elements, improving modifiability and reuse.

The state repository enables the various components of the application environment to share state and to communicate indirectly. This avoids duplication of data and allows different components to act in parallel.

The laws repository encapsulates the various laws as first–class elements in the agent system. This approach avoids that laws are scattered over different components of the system. On the other hand, explicitly modelling laws may induce a important computational overhead. If performance is a high-ranked quality, laws may be hard coded in the various applicable modules.

5 Component and Connector Communicating Processes View

The communicating processes view shows the multiagent system as a set of concurrently executing units and their interactions. The elements of the communicating processes view are *concurrent units*, *repositories*, and *connectors*. Concurrent units are an abstraction for more concrete software elements such as task, process, and thread. Connectors enable data exchange between concurrent units and control of concurrent units such as start, stop, synchronization, etc. The relationship in this view is *attachment* that indicates which connectors are connected to which concurrent units and repositories [13].

The communicating processes view explains which portions of the system operate in parallel and is therefore an important artefact to understand how the system works and to analyze the performance of the system. Furthermore, the view is important to decide which components should be assigned to which processes. Actually, we present the communicating processes view as a number of core components and overlay them with a set of concurrently executing units and their interactions.

The reference architecture provides one view packet of the component and connector communicating view. This view packet shows the main processes involved in perception, interaction, and communication in the situated multiagent system.

5.1 C & C Communicating Processes View Packet 1: Perception, Interaction, and Communication

Primary Presentation. The primary presentation is shown in Fig. 7.

Elements and their Properties. This view packet shows the main processes and repositories of agent and the application environment. We make a distinction between *active processes* that run autonomously, and *reactive processes* that are triggered by other processes to perform a particular task.

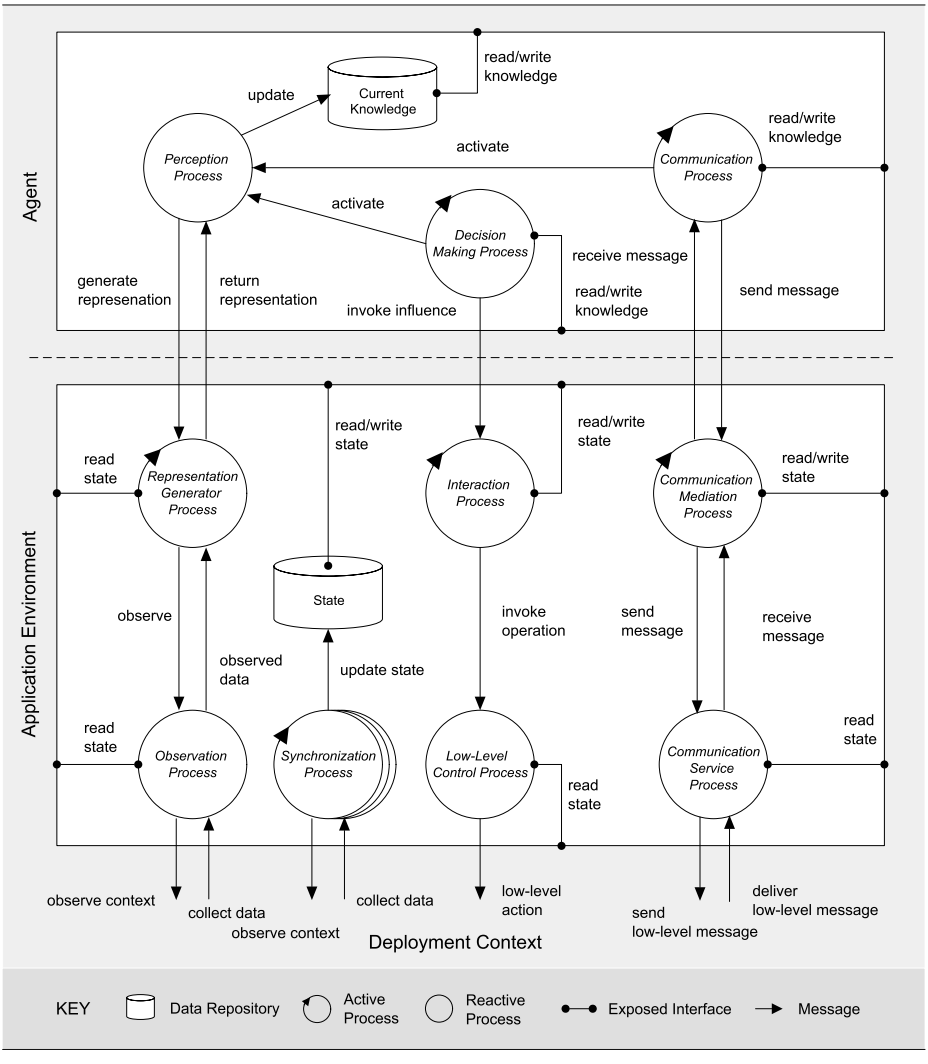

Fig. 7. Communicating processes view for perception, interaction, and communication

The discussion of the elements in this view packet is divided in four parts. Successively, we zoom in on the communicating processes of perception, interaction, and communication, and the synchronization processes of the application environment.

Perception. The `Perception Process` of agent is a reactive process that can be activated by the `Decision Making Process` and the `Communication Process`. Once activated, the perception process requests the `Representation Generator`

Process to generate a representation. The representation generator process collects the required state from the State repository of the application environment, and optionally it requests the Observation Process to collect additional data from the deployment context and possibly state of other nodes. State collection is subject to the perception laws. The observation process returns the observed data to the representation generator process, and subsequently the representation generator integrates the perceived state and generates a representation that is returned to the perception process of the agent. The perception process converts the representation to a percept that it uses to update the agent's Current Knowledge. Finally, the requesting process can read the updated state of the agent. The current knowledge repository can provide a notification mechanism to inform the decision making and communication process when a state update is completed.

Interaction. The Decision Making Process is an active process of agent that selects and invokes influences in the environment. The Interaction Process collects the concurrently invoked influences 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 executed by the interaction process, operations that attempt to modify state of the deployment context are forwarded to the Low-Level Control Process. This latter process converts the operations into low–level interactions in the deployment context.

Communication. The Communication Process is an active process that handles the communicative interactions of the agent. Newly composed messages are passed to the Communication Mediation Process that applies the communication laws and subsequently passes the messages to the Communication Service Process. This latter process converts the messages into low–level interactions that are transmitted via the deployment context. Furthermore, the Communication Service Process collects low–level messages from the deployment context, converts the messages into a format understandable for the agents, and forward the messages to the communication mediation process that delivers the messages to the communication process of the appropriate agent. Messages directed to agents that are located at the same host are directly transferred to the appropriate agents.

Synchronization Processes in the Application Environment. The Synchronization Processes are active processes that (1) monitor application specific parts of the deployment context and keep the corresponding state of the application environment up-to-date, (2) maintain application specific dynamics in the application environment, and (3) synchronize state among nodes according to the requirements of the application at hand.

Variation Mechanisms. There is one variation mechanism in this view packet.

CP1 *State Synchronization.* The parts of the deployment context for which a representation has to be maintained in the application environment have to

be defined. The deployment context may provide a notification mechanism to inform synchronization processes about changes, or the processes may poll the deployment context according to specific time schemes. Besides, for each activity in the application environment that happens independently of agents, an active process has to be defined. Finally, processes to synchronize state among nodes must be defined. Appropriate middleware may be used to support the synchronization of state among nodes.

Design Rationale. Agents are provided with two active processes, one for decision making and one for communication. This approach allows these processes to run in parallel, improving efficiency. Communication among the processes happens indirectly via the current knowledge repository. The perception process is reactive, the agent only senses the environment when required for decision making and communicative interaction. As such, the perception process is only activated when necessary.

The application environment is provided with separate processes to collect and process perception requests, handle influences, and provide message transfer. The observation process is reactive, it collects data from the deployment context when requested by the representation generator. The low-level control process is also reactive, it provides its services on command of the interaction mediation process. The communication service is reactive process that handles the transmission of messages when new messages arrives. Finally, synchronization processes are active processes that act largely independent of other processes in the system. Synchronization processes monitor particular dynamics in the deployment context and keep the corresponding representations up-to-date in the state of the application environment; they maintain dynamics in the application environment that happen independent of agents, and synchronize state among nodes.

Active processes represent loci of continuous activity in the system. By letting active processes run in parallel, different activities in the system can be handled concurrently, improving efficiency. Reactive processes, on the other hand, are only activated and occupy resources when necessary.

6 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 [45,2]. 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.

6.1 Multiagent System for the AGV Transportation System

In the project, we have applied a multiagent system 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 mediation level that agents can use as a medium 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].

6.2 Collision Avoidance

As an illustration of the software architecture of the AGV transportation system, we take a closer look at collision avoidance. AGV agents avoid collisions by coordinating with other agents through the virtual environment. AGV agents mark the path they are going to drive in their environment using hulls. The hull of an AGV demarcates the physical area the AGV occupies in the virtual environment. A series of hulls then describes the physical area an AGV occupies along a certain path. If the area is not marked by other hulls (the AGVs 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 virtual environment resolves the conflict taking into account the priorities of the transported loads to determine which AGV can move on. Afterwards, the AGV agent removes the markings in the virtual environment. Fig. 8 shows the primary presentation of the communicating processes view for collision avoidance. The communicating processes view presents the basic layers of the AGV control system and overlay them with the main processes and repositories involved in collision avoidance.

The top layer consists of the AGV agent that is responsible for controlling an AGV vehicle. The main functionalities of an AGV agent are: (1) obtaining

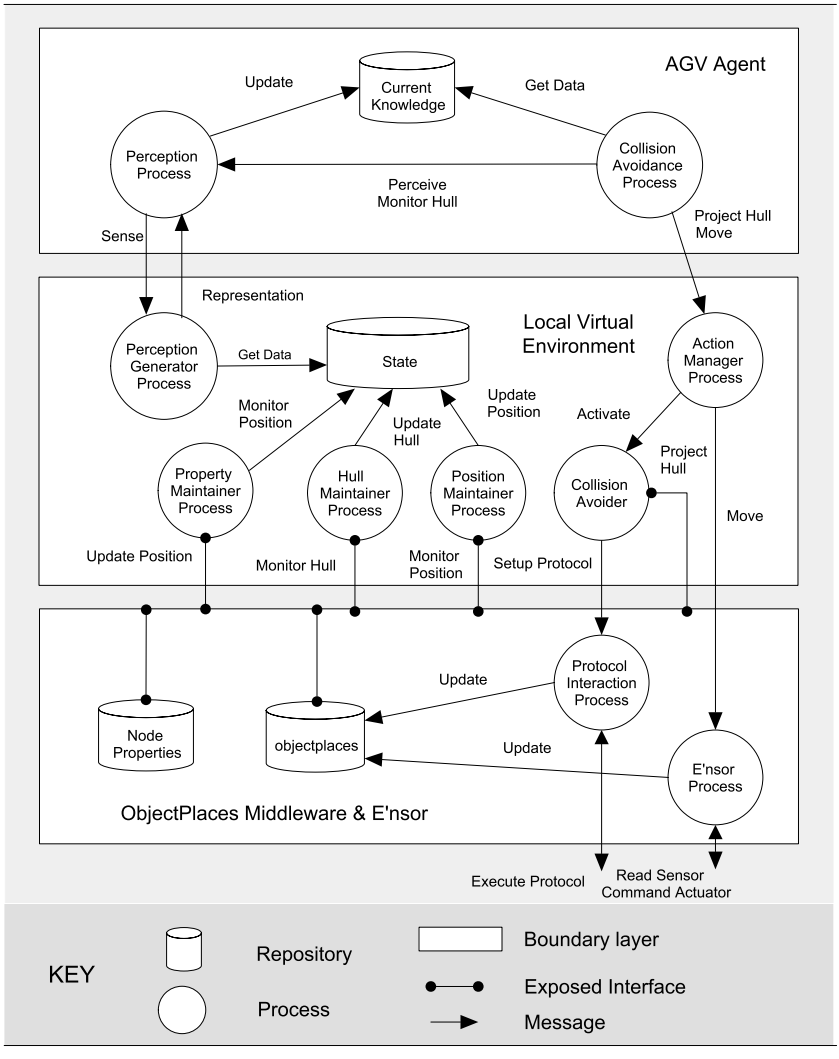

Fig. 8. Communicating processes for collision avoidance

transport tasks; (2) efficiently and safely handling jobs; (3) maintaining the AGV machine (charging battery, calibrating etc.).

The middle layer consists of the local virtual environment that is responsible for (1) representing and maintaining relevant state of the physical environment and the AGV vehicle; (2) representing additional state for coordination purposes; (3) synchronization of state with neighboring local virtual environments.

The bottom layer consists of the ObjectPlaces middleware and the E'nsor software. The ObjectPlaces middleware enables communication with software systems on other nodes, providing a means to synchronize the state of the local

virtual environment with the state of local virtual environments on neighboring nodes. E'nsor is the low-level control software of the AGV vehicle. The E'nsor software provides an interface to command the AGV vehicle and to read out its status. The E'nsor interface defines instructions to move the vehicle over a particular distance and possibly execute an action at the end of the trajectory such as picking up a load. The physical execution of the commands is managed by E'nsor. As such, the AGV agent can control the movement and actions of the AGV at a fairly high-level of abstraction.

We now discuss the main architectural elements involved in collision avoidance in turn.

The **Perception Process** is part of the agent's perception component, and corresponds to the perception process in the reference architecture. If the perception process receives a request for perception, it requests the up-to-date data from the local virtual environment and updates the agent's current knowledge.

The **Perception Generator Process** is part of the representation generator and corresponds to the representation generator process in the reference architecture. This process is responsible for handling perception requests, it derives the requested data from the state repository of the local virtual environment according to the given foci. An observation & data processing process (as in the the reference architecture) is absent in the local virtual environment. State from the deployment context and other nodes that is needed by the AGV agent is maintained by dedicated synchronization processes.

Collision Avoidance Process is part of the AGV agent's decision making component and is a helper process of the decision making process. The collision avoidance process calculates the required hull projection for collision avoidance, based on the most up-to-date data, and projects the hull in local virtual environment. Once the hull is locked, the collision avoidance process invokes a move command in the local virtual environment.

The **Action Manager Process** is part of the interaction component and corresponds to the interaction process in the reference architecture. The action manager process collects the influences invoked in the local virtual environment and dispatches them to the applicable processes. For a hull projection, the action manager process passes the influence to the collision avoider process of the local virtual environment. A move influence is passed to the E'nsor process.

Objectplaces repository is a repository of data objects in the ObjectPlaces middleware that contains the hulls the AGV agent has requested.

NodeProperties is a data repository in the middleware in which relevant properties of the node are maintained, an example is the AGV's current position. Maintenance of node properties in the repository is handled by the **Property Maintainer Process**. This process is a an instance of a synchronization process of the local virtual environment. The data objects of the NodeProperties repository are used by the middleware to synchronize the state among local virtual environment on neighboring nodes. For example, the current position in the

node properties repository is used by the ObjectPlaces middleware to determine whether the AGV is within collision range of other AGVs.

The **Collision Avoider** is a helper process of the action manager process that projects the requested hull in the objectplaces repository and initiates the collision avoidance protocol in the middleware.

The **Protocol Interaction Process** is a process of the ObjectPlaces middleware that is responsible for executing the mutual exclusion protocol for collision avoidance with the AGVs in collision range. This process maintains the state of the agent's hull in the objectplaces repository.

The **Hull Maintainer Process and Position Maintainer Process** are part of the synchronization component. These processes are application-specific instances of synchronization processes in the reference architecture. The hull maintainer process monitors the hull object in the objectplaces repository and keeps the state of the hull in the state repository of the local virtual environment consistent. The position maintainer process maintains in a similar way the actual position of the AGV vehicle.

Finally, the **E'nsor Process** is part of E'nsor and corresponds to a low-level control process in the reference architecture. The E'nsor process (1) periodically provides updates of the vehicles physical state (such as position and battery status), and (2) translates the high-level actions from the action manager process into low-level commands for the vehicle actuators.

7 Related Work

In this section, we discuss a number of representative reference architectures and reference models for multiagent systems.

7.1 PROSA: Reference Architecture for Manufacturing Systems

[49] defines a reference architecture as a set of coherent engineering and design principles used in a specific domain. PROSA—i.e. an acronym for Product–Resource–Order–Staff Architecture—defines a reference architecture for a family of coordination and control application, with manufacturing systems as the main domain. These systems are characterized by frequent changes and disturbances. PROSA aims to provide the required flexibility to cope with these dynamics.

The PROSA reference architecture [11,36] is built around three types of basic agents: resource agent, product agent, and order agent. A resource agent contains a production resource of the manufacturing system, and an information processing part that controls the resource. A product agent holds the know-how to make a product with sufficient quality, it contains up-to-date information on the product life cycle. Finally, an order agent represents a task in the manufacturing system, it is responsible for performing the assigned work correctly and on time. The agents exchange knowledge about the system, including process

knowledge (i.e. how to perform a certain process on a certain resource), production knowledge (i.e. how to produce a certain product using certain resources), and process execution knowledge (i.e. information and methods regarding the progress of executing processes on resources). Staff agents are supplementary agents that can assist the basic agents in performing their work. Staff agents allow to incorporate centralized services (e.g, a planner or a scheduler). However, staff agents only give *advice* to basic agents, they do not introduce rigidity in the system.

The PROSA reference architecture uses object-oriented concepts to model the agents and their relationships. Aggregation is used to represent a cluster of agents that in turn can represent an agent at a higher level of abstraction. Specialization is used to differentiate between the different kinds of resource agents, order agents, and product agents specific for the manufacturing system at hand.

The target domain of PROSA is a sub-domain of the target domain of the reference architecture for situated multiagent systems. As such, the PROSA reference architecture is more specific and tuned to its target domain. The specification of the PROSA reference architecture is descriptive. PROSA specifies the responsibilities of the various agent types in the system and their relationships, but abstracts from the internals of the agents. As a result, the reference architecture is easy to understand. Yet, the informal specification allows for different interpretations. An example is the use of object-oriented concepts to specify relationships between agents. Although intuitive, in essence it is unclear what the precise semantics is of notions such as "aggregation" and "specialization" for agents. What are the constraints imposed by such a hierarchy with respect to the behavior of agents as autonomous and adaptive entities? Without a rigorous definition, such concepts inevitable leads to confusion and misunderstanding.

[21] presents an interesting extension of PROSA in which the environment is exploited to obtain BDI (Believe, Desire, Intention [27]) functionality for the various PROSA agents. To avoid the complexity of BDI-based models and the accompanying computational load, the agents delegate the creation and maintenance of complex models of the environment and other agents to the environment. The approach introduces the concept of "delegate multiagent system". A delegate multiagent system consists of light-weight agents which can be issued by the different PROSA agents. These ant-like agents can explore the environment, bring relevant information back to their responsible agent, and put the intentions of the responsible agent as information in the environment. This allows delegate multiagent systems of different agents to coordinate by aligning or adapting the information in the environment according to their own tasks. A similar idea was proposed by Bruecker in [10], and has recently further been elaborated by Parunak and Brueckner, see [26]. The use of the environment in the work of [21] is closely connected to our perspective on the role of the environment as an exploitable design abstraction. The main challenge is now to develop an architecture that integrates the BDI functionality provided by a

delegate multiagent system with the architecture of the cognitive agent that issues the delegate multiagent system in the environment.

7.2 Aspect-Oriented Agent Architecture

In [17], Garcia et al. observe that several agent concerns such as autonomy, learning, and mobility crosscut each other and the basic functionality of the agent. 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.

The authors make a distinction between basic concerns of agent architectures, and additional concerns that are optional. Basic concerns are features that are incorporated by all agent architectures and include knowledge, interaction, adaptation, and autonomy. Examples of additional concerns are mobility, learning, and collaboration. 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 [24]. Each aspect modularizes a particular concern of the agent (basic and additional concerns). The architectural elements of the aspect-oriented agent architecture provide two types of interfaces: regular and crosscutting interfaces. A crosscutting interface specifies when and how an architectural aspect affects other architectural elements. The authors claim that the proposed approach provides a clean separation between the agent's basic functionality and the crosscutting agent properties. The resulting architecture is easier to understand and maintain, and improves reuse.

State-of-the-art research in aspect-oriented software development is mainly directed at the specification of aspects at the programming level, and this is the same for the work of Garcia and his colleagues. The approach has been developed bottom up, resulting in specifications of aspects at the architectural level that mirror aspect-oriented implementation techniques. The notion of crosscutting interface is a typical example. Unfortunately, a precise semantics of "when and how an architectural aspect affects other architectural elements" is lacking.

The aspect-oriented agent architecture applies a different kind of modularization as we did in the reference architecture for situated multiagent systems. Whereas a situated agent in the reference architecture is decomposed in functional building blocks, Garcia and his colleagues take another perspective on the decomposition of agents. The main motivation for the aspect-oriented agent architecture is to separate different concerns of agents aiming to improve understandability and maintenance. Yet, it is unclear whether the interaction of the different concerns in the kernel (feature interaction [12]) will not lead to similar problems the approach initially aimed to resolve. Anyway, crosscutting concerns in multiagent systems are hardly explored and provide an interesting venue for future research.

7.3 Architectural Blueprint for Autonomic Computing

Autonomic Computing is an initiative started by IBM in 2001. Its ultimate aim is to create self-managing computer systems to overcome their growing complexity [23]. IBM has developed an architectural blueprint for autonomic computing [1]. This architectural blueprint specifies the fundamental concepts and the architectural building blocks used to construct autonomic systems.

The blueprint architecture organizes an autonomic computing system into five layers. The lowest layer contains the system components that are managed by the autonomic system. System components can be any type of resource, a server, a database, a network, etc. The next layer incorporates touchpoints, i.e. standard manageability interfaces for accessing and controlling the managed resources. Layer three constitutes of autonomic managers that provide the core functionality for self-management. An autonomic manager is an agent-like component that manages other software or hardware components using a control loop. The control loop of the autonomic manager includes functions to monitor, analyze, plan and execute. Layer four contains autonomic managers that compose other autonomic managers. These composition enables system-wide autonomic capabilities. The top layer provides a common system management interface that enables a system administrator to enter high-level policies to specify the autonomic behavior of the system. The layers can obtain and share knowledge via knowledge sources, such as a registry, a dictionary, and a database.

We now briefly discuss the architecture of an autonomic manager, the most elaborated part in the specification of the architectural blueprint. An autonomic manager automates some management function according to the behavior defined by a management interface. Self-managing capabilities are accomplished by taking an appropriate action based on one or more situations that the autonomic manager senses in the environment. Four architectural elements provide this control loop: (1) the monitor function provides the mechanisms that collect, aggregate, and filter data collected from a managed resource; (2) the analyze function provides the mechanisms that correlate and model observed situations; (3) the plan function provides the mechanisms that construct the actions needed to achieve the objectives of the manager; and (4) the execute function provides the mechanisms that control the execution of a plan with considerations for dynamic updates. These four parts work together to provide the management functions of the autonomic manager.

Although presented as architecture, to our opinion, the blueprint describes a reference model. The discussion mainly focusses on functionality and relationships between functional entities. The specification of the horizontal interaction among autonomic managers is lacking in the model. Moreover, the functionality for self-management must be completely provided by the autonomic managers. Obviously, this results in complex internal structures and causes high computational loads.

The concept of application environment in the reference architecture for situated multiagent systems provides an interesting opportunity to manage complexity, yet, it is not part of the IBM blueprint. The application environment could enable the coordination among autonomic managers and provide supporting services. Laws embedded in the application environment could provide a means to impose rules on the autonomic system that go beyond individual autonomic managers.

7.4 A Reference Model for Multiagent Systems

In [25], Modi et al. present a reference model for agent-based systems. The aim of the model is fourfold: (1) to establish a taxonomy of concepts and definitions needed to compare agent-based systems; (2) to identify functional elements that are common in agent-based systems; (3) to capture data flow dependencies among the functional elements; and (4) to specify assumptions and requirements regarding the dependencies among the elements.

The model is derived from the results of a thorough study of existing agent-based systems, including Cougaar [19], Jade [7], and Retsina [34]. The authors used reverse engineering techniques to perform an analysis of the software systems. Static analysis was used to study the source code of the software, and dynamic analysis to inspect the system during execution. Key functions identified are directory services, messaging, mobility, inter-operability services, etc.

Starting from this data a preliminary reference model was derived for agent-based systems. The authors describe the reference model by means of a layered view and a functional view. The layered view is comprised of agents and their supporting framework and infrastructure which provide services and operating context to the agents. The model defines framework, platform, and host layers, which mediate between agents and the external environment. The functional view presents a set of functional concepts of agent-based systems. Example functionalities are administration (instantiate agents, allocate resources to agents, terminate agents), security (prevent execution of undesirable actions by entities from within or outside the agent system), conflict management (facilitate and enable the management of interdependencies between agents activities), and messaging (enable information exchange between agents).

The reference model in an interesting effort towards maturing the domain. In particular, the reference model aims to be generic but does not make any recommendation about how to best engineer an agent-based system. Putting the focus on abstractions helps to resolve confusion in the domain and facilitates acquisition of agent technology in practice.

Yet, since the authors have investigated only systems in which agents communicate through message exchange, the resulting reference model is biased towards this kind of agent systems. The concept of environment as a means for information sharing and indirect coordination of agents is absent. On the other hand, it is questionable whether developing one common reference model for the broad family of agent-based system is desirable.

8 Conclusions

In this paper, we presented a reference architecture for situated multiagent systems. The general goal of the reference architecture is to support the architectural design of self-managing applications. Concrete contributions are: (1) the reference architecture defines how various mechanisms of adaptivity for situated multiagent systems are integrated in one architecture; (2) the reference architecture provides a blueprint for architectural design, it facilitates deriving new software architectures for systems that share its common base; and (3) the reference architecture reifies the knowledge and expertise we have acquired in our research, it offers a vehicle to study and learn the advanced perspective on situated multiagent systems we have developed in our research.

We presented the reference architecture by means of three views that describe the architecture from different perspectives. Views are presented as a number of view packets. A view packet focusses on a particular part of the reference architecture. We gave a primary presentation of each view packet and we explained the properties of the architectural elements. Besides, each view packet is provided with a number of variation mechanisms and a design rationale. Variation mechanisms describe how the view packet can be applied to build concrete software architectures. The design rationale explains the underlying design choices of the view packet and the quality attributes associated with the various view packets. [37] provides a detailed formal specification of the various architectural elements.

We illustrated how we have used the reference for the architectural design of an AGV transportation system. In particular, we showed how a set of abstractly defined processes in the reference architecture are instantiated to provide the functionality for collision avoidance.

The reference architecture serves as a blueprint for developing concrete software architectures. It integrates a set of architectural patterns architects can draw from during architectural design. However, the reference architecture is not a ready-made cookbook for architectural design. It offers a set of reusable architectural solutions to build software architectures for concrete applications. Yet, applying the reference architecture does not relieve the architect from difficult architectural issues, including the selection of supplementary architectural approaches to deal with specific system requirements. We consider the reference architecture as a *guidance* for architectural design that offers a reusable set of architectural assets for building software architectures for concrete applications. Yet, this set is not complete and needs to be complemented with additional architectural approaches.

References

1. IBM, An Architectural Blueprint for Autonomic Computing, (6/2006). www-03.ibm.com/autonomic/.
2. 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/.

3. The Unified Modeling Language, (8/2006). http://www.uml.org/.
4. C. Atkinson and T. Kuhne. Aspect-Oriented Development with Stratified Frameworks. *IEEE Software*, 20(1):81–89, 2003.
5. J. Austin. *How To Do Things With Words*. Oxford University Press, Oxford, UK, 1962.
6. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley Publishing Comp., 2003.
7. F. Bellifemine, A. Poggi, and G. Rimassa. Jade, A FIPA-compliant Agent Framework. In *4th International Conference on Practical Application of Intelligent Agents and Multi-Agent Technology*, London, UK, 1999.
8. N. Boucké, D. Weyns, T. Holvoet, and K. Mertens. Decentralized allocation of tasks with delayed commnencement. In *2nd European Workshop on Multi-Agent Systems, EUMAS*, Barcelona, Spain, 2004.
9. N. Boucké, D. Weyns, K. Schelfthout, and T. Holvoet. Applying the ATAM to an Architecture for Decentralized Contol of a AGV Transportation System. In *2nd International Conference on Quality of Software Architecture, QoSA*, Vasteras, Sweden, 2006. Springer.
10. S. Brueckner. *Return from the Ant, Synthetic Ecosystems for Manufacturing Control*. Ph.D Dissertation, Humboldt University, Berlin, Germany, 2000.
11. H. Van Brussel, J. Wyns, P. Valckenaers, L. Bongaerts, and P. Peeters. Reference Architecture for Holonic Manufacturing Systems: PROSA. *Jounal of Manufactoring Systems*, 37(3):255–274, 1998.
12. M. Calder, M. Kolberg, E. Magill, and S. Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41(1):115–141, 2003.
13. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison Wesley Publishing Comp., 2002.
14. C. Cuesta, M. del Pilar Romay, P. de la Fuente, and M. Barrio-Solofzano. Architectural Aspects of Architectural Aspects. In *2nd European Wirkshop on Software Architecture, EWSA*, Lecture Notes in Computer Science, Vol. 3527. Springer, 2005.
15. J. Ferber and J. Muller. Influences and Reaction: a Model of Situated Multiagent Systems. *2nd International Conference on Multi-agent Systems, Japan, AAAI Press*, 1996.
16. FIPA. Foundation for Intelligent Physical Agents, FIPA Abstract Architecture Specification. http://www.fipa.org/repository/bysubject.html, (8/2006).
17. A. Garcia, U. Kulesza, and C. Lucena. Aspectizing Multi-Agent Systems: From Architecture to Implementation. In *Software Engineering for Multi-Agent Systems III, SELMAS 2004*, Lecture Notes in Computer Science, Vol. 3390. Springer, 2005.
18. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 3(8):231–274, 1987.
19. A. Helsinger, R. Lazarus, W. Wright, and J. Zinky. Tools and Techniques for Performance Measurement of Large Distributed Multiagent Systems. In *2nd International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS, Melbourne, Victoria, Australia*. ACM, 2003.
20. A. Holvoet. Visualisation of a Peer-to-Peer Network. *Master Thesis, Katholieke Universiteit Leuven, Belgium*, 2004.
21. T. Holvoet and P. Valckenaers. Exploiting the Environment for Coordinating Agent Intentions. In *3th International Workshop on Environments for Multiagent Systems, E4MAS*, Hakodate, Japan, 2006.

22. E. Kendall and C. Jiang. Multiagent System Design Based on Object Oriented Patterns. *Journal of Object Oriented Programming*, 10(3):41–47, 1997.
23. J. Kephart and D. Chess. The Vision of Autonomic Computing. *IEEE Computer Magazine*, 36(1).
24. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Vol. 1241, Berlin, Heidelberg, New York, 1997. Springer-Verlag.
25. P. Modi, S. Mancoridis, W. Mongan, W. Regli, and I. Mayk. Towards a Reference Model for Agent-Based Systems. In *Industry Track of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*, Hakodate, Japan, 2006. ACM.
26. H. V. D. Parunak and S. Brueckner. Concurrent Modeling of Alternative Worlds with Polyagents. In *7th International Workshop on Multi-Agent-Based Simulation*, Hakodate, Japan, 2006.
27. A. Rao and M. Georgeff. BDI Agents: From Theory to Practice. In *1st International Conference on Multiagent Systems, 1995, Agents, San Francisco, California, USA*. The MIT Press, 1995.
28. P. Reed. Reference Architecture: The Best of Best Practices. *The Rational Edge*, 2002. www-128.ibm.com/developerworks/rational/library/2774.html.
29. K. Schelfthout and T. Holvoet. Views: Customizable abstractions for context-aware applications in MANETs. *Software Engineering for Large-Scale Multi-Agent Systems, St. Louis, USA*, 2005.
30. K. Schelfthout, D. Weyns, and T. Holvoet. Middleware that Enables Protocol-Based Coordination Applied in Automatic Guided Vehicle Control. *IEEE Distributed Systems Online*, 7(8), 2006.
31. W. Schols, T. Holvoet, N. Boucké, and D. Weyns. Gradient Field Based Transport Assignment in AGV Systems. In *CW-425, Technical Report*. Departement of Computer Science, Katholieke Universiteit Leuven, Belgium. http://www.cs.kuleuven.ac.be/publicaties/rapporten/CW/2005/.
32. E. Steegmans, D. Weyns, T. Holvoet, and Y. Berbers. A Design Process for Adaptive Behavior of Situated Agents. In *Agent-Oriented Software Engineering V, 5th International Workshop, AOSE, New York, NY, USA*, Lecture Notes in Computer Science, Vol. 3382. Springer, 2004.
33. E. Steegmans, D. Weyns, T. Holvoet, and Y. Berbers. Designing Roles for Situated Agents. In *5th International Workshop on Agent-Oriented Software Engineering*, New York, NY, USA, 2004.
34. K. Sycara, M. Paolucci, M. Van Velsen, and J. Giampapa. The RETSINA MAS Infrastructure. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):29–48, 2003.
35. B. Tekinerdogan. ASAAM: Aspectual Software Architecture Analysis Method. In *4th Working Conference on Software Architecture, WICSA, Oslo, Norway*. IEEE Computer Society, 2004.
36. P. Valckenaers and H. Van Brussel. Holonic Manufacturing Execution Systems. *CIRP Annals-Manufacturing Technology*, 54(1):427–432, 2005.
37. D. Weyns. An Architecture-Centric Approach for Software Engineering with Situated Multiagent Systems. *Ph.D Dissertation: Katholieke Universiteit Leuven*, 2006.
38. D. Weyns, N. Boucké, and T. Holvoet. Gradient Field Based Transport Assignment in AGV Systems. In *5th International Joint Conference on Autonomous Agents and Multi-Agent Systems, AAMAS, Hakodate, Japan*, 2006.

39. D. Weyns, A. Helleboogh, and T. Holvoet. The Packet-World: a Test Bed for Investigating Situated Multi-Agent Systems. In *Software agent-based applications, platforms, and development kits*. Whitestein Series in Software Agent Technology, 2005.
40. D. Weyns and T. Holvoet. Model for Simultaneous Actions in Situated Multiagent Systems. In *Multiagent System Technologies, 1st German Conference, MATES 2003, Erfurt, Germany*, Lecture Notes in Computer Science, Vol. 2831. Springer Verlag, 2003.
41. D. Weyns and T. Holvoet. Formal Model for Situated Multi-Agent Systems. *Fundamenta Informaticae*, 63(1-2):125–158, 2004.
42. D. Weyns and T. Holvoet. Regional Synchronization for Situated Multi-agent Systems. In *3th International Central and Eastern European Conference on Multi-Agent Systems, Prague, Czech Republic*, Lecture Notes in Computer Science, Vol. 2691. Springer Verlag, 2004.
43. D. Weyns and T. Holvoet. Architectural Design of an Industrial AGV Transportation System with a Multiagent System Approach. In *Software Architecture Technology User Network Workshop, SATURN*, Pittsburg, USA, 2006. Software Engineering Institute, Carnegie Mellon University.
44. D. Weyns, K. Schelfthout, T. Holvoet, and O. Glorieux. Towards Adaptive Role Selection for Behavior-Based Agents. In *Adaptive Agents and Multi-Agent Systems II: Adaptation and Multi-Agent Learning*, Lecture Notes in Computer Science, Vol. 3394. Springer, 2005.
45. D. Weyns, K. Schelfthout, T. Holvoet, and T. Lefever. Decentralized control of E'GV transportation systems. In *4th Joint Conference on Autonomous Agents and Multiagent Systems, Industry Track*, Utrecht, The Netherlands, 2005. ACM Press, New York, NY, USA.
46. D. Weyns, E. Steegmans, and T. Holvoet. Integrating Free-Flow Architectures with Role Models Based on Statecharts. In *Software Engineering for Multi-Agent Systems III, SELMAS*, Lecture Notes in Computer Science, Vol. 3390. Springer, 2004.
47. D. Weyns, E. Steegmans, and T. Holvoet. Protocol Based Communication for Situated Multi-Agent Systems. In *3th Joint Conference on Autonomous Agents and Multi-Agent Systems*, New York, USA, 2004. IEEE Computer Society.
48. D. Weyns, E. Steegmans, and T. Holvoet. Towards Active Perception in Situated Multi-Agent Systems. *Applied Artificial Intelligence*, 18(9-10):867–883, 2004.
49. J. Wyns, H. Van Brussel, P. Valckenaers, and L. Bongaerts. Workstation Architecture in Holonic Manufacturing Systems. In *28th CIRP International Seminar on Manufacturing Systems*, Johannesburg, South Africa, 1996.