KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

An Architecture-Centric Approach for Software Engineering with Situated Multiagent Systems

Jury :
Prof. Dr. ir.-arch. H. Neuckermans, voorzitter
Prof. Dr. T. Holvoet, promotor
Prof. Dr. ir. P. Verbaeten, promotor
Prof. Dr. ir. Y. Berbers
Prof. Dr. ir. H. Blockeel
Prof. Dr. ir. W. Joosen
Dr. H.V.D. Parunak
(NewVectors, LLC, Ann Arbor, MI, USA)
Prof. Dr. F. Zambonelli
(University of Modena and Reggio Emilia, Italy)

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de Ingenieurswetenschappen

door

Danny Weyns

U.D.C. 681.3, 681.3*D2

October 2006

D/2006/7515/66
ISBN 90–5682–732–4

Abstract

Developing and managing today's distributed applications is hard. Three important reasons for the increasing complexity that characterize a large family of systems are: (1) stakeholders involved in the systems have various, often conflicting quality requirements; (2) the systems are subject to highly dynamic and changing operating conditions; (3) activity in the systems is inherently localized, global control is hard to achieve or even impossible.

In this dissertation, we present an approach for developing such complex systems. The approach integrates situated multiagent systems as software architecture in a mainstream software engineering process. Key aspects of the approach are architecture-centric software development, self-management, and decentralized control. Architecture-centric software development compels the stakeholders involved in a system to deal explicitly with quality goals and tradeoffs between the various system requirements. Self-management enables a software system to deal autonomously with the dynamic and changing circumstances in which it has to operate. Key qualities for endowing systems with abilities to manage dynamism and change are flexibility and openness. Decentralized control is essential to cope with the inherent locality of activity. In a system where global control is not an option, the functionality of the system has to be achieved by collaborating subsystems.

We present an advanced model for situated multiagent systems that integrates the environment as a first-class design abstraction with an integral model for situated agents that provides advanced mechanisms for adaptive behavior. These mechanisms enable situated agents to manage the changing situation in the environment autonomously; the multiagent system can cope with agents leaving the system and new agents that enter. Control in a situated multiagent system is decentralized, situated agents cooperate to achieve the overall functionality of the system.

From our experiences with building various situated multiagent system applications, we have developed a reference architecture for situated multiagent systems. This reference architecture maps the advanced model for situated multiagent systems on an abstract system decomposition. We give an overview of the various views of the architecture, and we explain how the reference architecture can guide architects when developing new applications that share the common base of the reference architecture.

We have applied a situated multiagent system in a industrial automated transportation system. The architectural design, the development, and the evaluation of this complex application has considerably contributed to the development of the reference architecture. We give an overview of the software architecture of the system, and we discuss the evaluation of the architecture. The successful development of this challenging application demonstrates how multiagent systems can be integrated as software architecture in mainstream software engineering.

Acknowledgements

Deze thesis is het resultaat van het onderzoek dat ik de voorbije vijf jaren heb verricht in de DistriNet onderzoeksgroep. Het was een fascinerende en uitermate leerzame ervaring. Ik heb het geluk gehad in een echt team te kunnen werken—AgentWise, met een uitstekende coach. Het onderzoek heeft me naar verre plaatsen op de wereld gebracht, ik heb interessante mensen ontmoet en samen met hen heb ik fascinerende dingen op touw gezet. Bij het afsluiten van mijn doctoraat wil ik iedereen die dit mee heeft mogelijk gemaakt van harte bedanken.

Vooreerst wil ik mijn promotoren Prof. Tom Holvoet en Prof. Pierre Verbaeten van harte bedanken voor het vertrouwen dat ze mij hebben geschonken om mijn onderzoek te kunnen verrichten. Bedankt voor de ruimte Tom die ik heb gekregen om mijn eigen weg te kunnen zoeken. Bedankt voor je luisterbereidheid, ondanks alle drukte. En bovenal bedankt voor je scherpe feedback, het heeft me telkens opnieuw gestimuleerd om het beste uit mezelf naar boven te brengen.

Naast Tom en Pierre bedank ik de leden van mijn begeleidingscommissie Prof. Yolande Berbers en Prof. Hendrik Blockeel voor het zorgvuldig nalezen van deze tekst en de waardevolle commentaar. I would like to express my sincere gratitude to Dr. Van Parunak, Prof. Franco Zambonelli, and Prof. Wouter Joosen for accepting to be members of the jury, and to Prof. Herman Neuckermans for chairing the jury.

Thanks to Liz Sonenberg, Wouter Joosen, and Pieter Jan 't Hoen for the valuable feedback on the initial outline of this thesis.

Een bijzonder woord van dank aan Eddy Truyen. Als begeleider van mijn master thesis en daarna als bureaugenoot heeft hij mij naadloos in het onderzoek ingeleid. Hij heeft me de principes—en de passie—bijgebracht om papers te schrijven. Eddy, ik denk met bijzonder veel genoegen terug aan onze jaren in de M-blok.

Zonder mijn collega's van AgentWise kan ik me mijn onderzoek moeilijk voorstellen. De kritische houding voor mekaars werk heeft ons telkens opnieuw de grenzen doen verleggen. Mijn bijzondere dank gaat uit naar Elke Steegmans voor onze samenwerking rond rollen, en naar Kurt Schelfthout met wie ik lief en leed heb

gedeeld bij de ontwikkeling van de controle software voor automatisch bestuurde voertuigen op Egemin. Ik bedank Alexander Helleboogh voor de aangename samenwerking. Bedankt Lexe voor al onze "stimulerende coffiepauzes" waaruit menig nieuw inzicht is ontsproten. Dank ook aan Nelis Boucké voor de samenwerking. Nelis, je creativiteit en kritische kijk hebben me altijd plezier gedaan, ik kijk uit naar onze verdere samenwerking. Ik wil iedereen van het team ook bedanken voor de inzet voor de E4MAS workshops.

Graag wil ook de onderzoekers van DistriNet bedanken voor de aandacht en de stimulerende feedback op mijn onderzoek tijdens de voorbije jaren. Dank ook aan Esther, Karin en Margot, alsook Jean en de rest van de systeemgroep voor jullie inzet.

Het EMC^2 project in samenwerking met Egemin was een bijzonder leerrijke ervaring. Dank in het bijzonder aan Tom Lefever en Jan Wielemans. Tom en Jan, jullie engagement en inzet waren onontbeerlijk voor het succesvol toepassen van ons onderzoek in de praktijk.

In the autumn of 2003, I contacted Van Parunak and Fabien Michel with the idea to organize a workshop on the role of the environment in multiagent systems. Their enthusiastic reaction was the forerunner of an exciting experience. The successful E4MAS workshops in New York, Utrecht, and Hakodate have put the environment on the map of multiagent system research. Van and Fabien, thank you for the very enjoyable and productive collaboration! I am grateful to Andrea Omicini and Jim Odell for our collaboration, in particular the joint work on the JAAMAS paper. Thanks also to Alessandro Ricci, Mirko Viroli, Giuseppe Vizarri, Michael Schumacher, and Eric Platon, for the collaborations.

Ik wil mijn familie bedanken, in Testelt en in Lint. Va en moe, bedankt voor alle goede zorgen en de kansen die ik heb gekregen. Doctoraatswerk is boeiend, maar soms ook moeilijk. Omringd zijn door mensen die "mee" zijn in het avontuur is van ontschatbare waarde. Frankie, onze vrijdagavonden waren telkens een moment waar ik naar uit kon zien. Ik hoop dat we nog vele inspirerende gesprekken bij "een Roche" mogen beleven!

Lieve Marina, beter dan wie ken jij de grillen van mijn gedrevenheid. Zonder jouw begrip, je steun, je zorgzaamheid,... zou dit doctoraat nooit gelukt zijn. Lieve Eva en Tessa, eindelijk, papa is afgestudeerd! Jullie interesse voor mijn "agentjes" heeft me steeds plezier gedaan. Het is fijn te zien hoe jullie nu zelf met inzet en plezier studeren.

Danny Weyns
Augustus 2006

Contents

List of Figures

Chapter 1

Introduction

Developing and managing today's distributed applications is hard. Three important reasons for the complexity form the starting point of this research. One reason for the complexity is the increasing demand on the quality of software [35, 171]. Different stakeholders interested in the software (users, project leaders, architects, developers, maintainers, etc.) usually have various expectations on the quality of the software (performance, flexibility, maintainability, etc.). Dealing with these—usually competing—quality requirements, and the project and business constraints (budgets, schedules, etc.), pose difficult challenges to software engineers.

A second important reason why the development and management of distributed applications is hard are the highly dynamic and changing operating conditions in which today's distributed applications operate. Applications are expected to cope with dynamically changing workloads, unpredictable peaks in demand, continuous changes in availability of resources and services, particular failure modes, variations in network traffic, etc. Wireless communication introduces great opportunities for the further integration of computing resources, however, it also introduces additional complexity due to the pervasiveness of network connectivity and continuous topological changes in the network [195, 167, 242].

In addition to the complexity resulting from the demanding quality attributes and the dynamic operating conditions, the nature of a large family of such distributed applications is such that global control is hard to achieve. Activity in these applications is inherently localized, i.e. global access to resources is difficult to achieve or even infeasible. Example domains are automated transportation systems and robotics, mobile and ad-hoc networks, and wireless sensor networks.

In this research we propose an *architecture-centric* approach to develop such complex applications. Architecture, in particular software architecture, is crucial for managing complexity and achieving the required quality attributes of the system. The approach aims for *self-managing* systems, i.e. systems that are able to manage dynamism and change autonomously. The cornerstone of the approach is

situated multiagent systems. Situated multiagent systems provide a way to model self-managing decentralized systems; *decentralized control* is essential to cope with the inherent locality of activity of the target applications.

In the remainder of this introduction, we start by explaining self-management. Next, we introduce situated multiagent systems. We explain our perspective on software engineering with multiagent systems, and we give an overview of mechanisms for adaptation in situated multiagent systems we have developed. The introduction concludes with a summary of the contributions of this research and an overview of the text.

1.1 Self-Managing Applications

Information Technology industry recognizes the rising problems of managing complex distributed applications and has started initiatives to face them. Example are IBM's Autonomic Computing [2] and Microsoft's Dynamic Systems Initiative [4]. These and similar initiatives recognize *self-management* as the only option to face growing problems. The general idea of self-management is to endow computing systems with the ability to manage themselves according to high-level objectives specified by humans. [110] divides self-management into four functional areas: (1) self-configuration: i.e. automatically configure components to adapt themselves to different environments; (2) self-healing: i.e. automatically discover, diagnose, and correct faults; (3) self-optimization: i.e. automatically monitor and adapt resources to ensure optimal functioning regarding the defined requirements; and (4) self-protection: i.e. anticipate, identify, and protect against arbitrary attacks.

To enable self-management, distributed applications must be equipped with suitable techniques and supported by appropriate infrastructure. In this, the environment will occupy a prominent position. A self-managing system must take the appropriate actions based on the situation sensed in the environment. This requires functionality for monitoring, decision making, and action execution. Local adjustments have to be in line with system-wide self-managing policies. This requires coordination of behavior within and among different applications. To meet these challenges, a whole range of useful techniques have been explored, such as techniques for problem probing [48], learning and adaptation techniques [231, 151], market mechanisms [103], and techniques to control and exploit emergent behavior [238, 155]. Supporting middleware has been developed, see e.g. [140, 131, 86]. However, the ultimate realization of self-management poses enormous research challenges [109].

In our research, we consider self-management as a system's ability to manage dynamism and change autonomously. With dynamism and change we refer to the variable circumstances a system can be subjected to during operation, such as altering workloads, variations in availability of resources and services, and subsystems that join and leave. Our focus on self-management is closely related to

self-optimization and self-healing (as defined in [110]). Self-configuration and self-protection are outside the scope of this research.

1.2 Situated Multiagent Systems

Research in multiagent systems is concerned with the study, behavior, and construction of a collection of autonomous agents that interact with each other and their environment [188]. [72] defines a multiagent system as "a loosely coupled network of problem solvers (agents) that interact to solve problems that are beyond the individual capabilities or knowledge of each problem solver." Characteristics of multiagent systems are: (1) each agent has incomplete information or capabilities for solving the problem and, thus, has a limited viewpoint; (2) there is no system global control; (3) data are distributed; and (4) computation is asynchronous.

Situated multiagent systems are one family of multiagent systems. A situated multiagent system consists of a (distributed) environment populated with a set of agents that cooperate to solve a complex problem in a decentralized way. A situated agent has an explicit position in the environment and has local access to the environment, i.e. each agent is placed in a local context which it can perceive and in which it can act and interact with other agents. System functionality results from the agents' interactions in the environment, rather than from their individual capabilities.

A situated agent uses a behavior-based action selection mechanism to select actions. Behavior-based action selection is driven by stimuli perceived in the environment as well as internal stimuli. Situated agents employ internal state for decision making insofar this state relates to (1) general static information of the system (e.g. fixed priority rules); (2) dynamic information related to the agent's current context (e.g. a temporal agreement for collaboration with a neighboring agent); or (3) issues internal to the agent (e.g. a threshold value used as a switch for changing roles).

In situated multiagent systems, the environment is an essential part of the system. The environment encapsulates resources and enables agents to access the resources. Moreover, the environment enables agents to interact with one another, it is the medium for agents to share information and coordinate their behavior. A typical example of environment-mediated coordination is a digital pheromone [41, 52, 154] which software agents use to coordinate their behavior. A digital pheromone is a dynamic structure in the environment that aggregates with additional pheromone that is dropped, diffuses in space and evaporates over time. Agents can use pheromones to dynamically form paths to locations of interest. Indirect exchange of information through the environment enables the coordination of the agents' individual loci of control. Control in a situated multiagent system is thus distributed among agents and the mediating environment.

Self-Management. In a situated multiagent system, self-management is essentially based on the ability of agents to adapt their behavior. Due to the efficiency of action selection, situated agents can rapidly respond to changing circumstances. Besides this form of "instant" flexibility, specific mechanisms have been developed that allow situated agents to adapt their behavior over time. Mechanisms for adaptive behavior enable situated agents to switch to the most appropriate behavior when circumstances in the environment change over time. The environment can play an active role in self-management. An example of this latter is a pheromone path that disappears over time, preventing agents to follow paths to outdated information.

1.3 Multiagent Systems and Software Architecture

Our research puts forward an architecture-centric approach for software engineering with multiagent systems. Our perspective on the essential purpose of multiagent systems is as follows:

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

This perspective states that multiagent systems provides the software to *solve* a problem. In particular, a multiagent system *structures* the system as a number of *interacting elements* in order to achieve the *requirements* of the system. This is exactly what *software architecture* is about. [35] defines software architecture as: "the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." Software elements (or in general architectural elements) provide the functionality of the system, while the required quality attributes (performance, usability, modifiability, etc.) are primarily achieved through the structures of the software architecture.

As such, multiagent systems are in essence a family—yet a large family—of software architectures. Based on the problem analysis that yields the functional and quality attribute requirements of the system, the architect may or may not choose for a multiagent system-based solution. Quality attribute requirements such as flexibility, openness, and robustness may be arguments for the designer to choose for a multiagent system software architecture. As such, we consider multiagent systems as one valuable family of approaches to solve software problems in a large spectrum of possible ways to solve problems. Typical architectural elements of a multiagent system software architecture are agents, environment,

resources, services, etc. The relationships between the elements are very diverse, ranging from environment-mediated interaction between cooperative agents via virtual pheromone trails to complex negotiation protocols in a society of self-interested agents. In short, multiagent systems are a rich family of architectural approaches with specific characteristics, useful for a diversity of challenging application domains.

Reference Architecture for Situated Multiagent Systems. [78] defines a reference architecture as "the generalized architecture of several end systems that share a common domain." A reference architecture provides a base for instantiating new software architectures for classes of systems that share the common base of the reference architecture.

In our research we have developed a reference architecture for situated multiagent systems. The reference architecture embodies the knowledge and expertise we have acquired during our research. The reference architecture generalizes and extracts common functions and structures from various experimental applications we have studied and built. These applications provide different degrees of complexity and various forms of dynamism and change. Applications include a prototypical peer-to-peer file sharing system, a simulation of an automatic guided vehicle transportation system, and several experimental robotic applications. Besides these basic applications, the development of the reference architecture is considerably based on experiences with an industrial logistic transportation system for warehouses.

The reference architecture is an asset for reuse, it can serve as a blueprint for developing software architectures for the kind of distributed applications we target in this research. The reference architecture also offers a vehicle to study and learn the basic structures and mechanisms of situated multiagent systems.

Current Practice in Agent-Oriented Software Engineering. Whereas our research aims to integrate multiagent systems with mainstream software engineering, current practice in agent-oriented software engineering considers multiagent systems as a *radically new* way of engineering software. Here is a list of illustrating quotes from literature of the past five years:

- There is a fundamental mismatch between the concepts used by object-oriented developers and other mainstream software engineering paradigms, and the agent-oriented view. [...] Existing software development techniques are unsuitable to realize the potential of agents as a software engineering paradigm. [235]
- Whether agent-oriented approaches catch on as a software engineering paradigm [is determined by] the degree to which agents represent a radical departure from current software engineering thinking. [102]

- We are on the edge of a revolutionary shift of paradigm, pioneered by the multiagent systems community, and likely to change our very attitudes in software modelling and engineering. [242]

- Agent-based computing can be considered as a new general-purpose paradigm for software development, which tends to radically influence the way a software system is conceived and developed. [241]

This vision has led to the development of numerous multiagent system methodologies. Some of the methodologies focus on particular phases of the software development process, e.g. Gaia [240]. Others cover the full software development life cycle, an example is Tropos [81]. Some of the proposed methodologies adopt mechanisms and practices from mainstream software engineering. Prometheus [146] is inspired by object-oriented mechanisms. Mase [233] uses practices of the Unified Process. Adelfe [40] uses constructs of the Unified Modeling Language. However, nearly all methodologies take an independent position, little or not related to mainstream software engineering practice.

The firm position of being a radically new paradigm for software development isolates agent-oriented software engineering from mainstream software engineering. Instead of considering multiagent systems as a radically new approach for software development, we aim to integrate multiagent systems in a general software engineering process. By considering multiagent systems essentially as software architecture, multiagent systems get a clear and prominent role in the software development process. The architecture-centric approach for software engineering with multiagent systems proposed in this research contributes to the integration of multiagent systems with mainstream software engineering.

1.4 Mechanisms for Adaptivity in Situated Multiagent Systems

To support architectural design of self-managing applications with situated multiagent systems, we have developed an advanced model for situated multiagent systems that extends state-of-the-art approaches in the domain. This model integrates the environment as a first-class abstraction with an advanced model for situated agents that includes explicit support for different concerns of agents and that supports various mechanisms for adaptivity.

Environment as a First-Class Abstraction in Multiagent Systems. Our research puts forward the environment as an independent building block with state and processes that can be exploited when designing multiagent system applications. Important responsibilities of the environment are: (1) the environment enables agents to access resources external to the multiagent system, it provides a

medium for agents to share information, and it enables agents to interact; (2) the environment can mediate the access to resources and information, and the interaction among agents. As a mediator, the environment can define various laws for the multiagent system that constrain the activity of the agents; (3) the environment can maintain dynamics that happen independently of agents' activity, such as maintenance of digital pheromones and monitoring and maintaining a representation of resources external to the multiagent system.

Integral Model for Situated Agents. Existing models of situated agents are typically restricted to mechanisms for action selection. Other concerns such as perception and communication are not dealt with, or integrated in the action selection model in an ad-hoc manner. In our research, we have developed an advanced model for situated agents that integrates different concerns of agents and that supports various mechanism for adaptivity.

We have developed a model for *selective perception* that enables an agent to direct its perception at the relevant aspects of the environment according to its current tasks. Selective perception allows an agent to adapt its observation with changing circumstances. This facilitates better situation awareness and helps to keep processing of perceived data under control.

We have endowed situated agents with abilities for explicit social interaction. This enables agents to exchange information directly with one another and set up collaborations. In particular, we have extended behavior-based action selection mechanisms with the notions of *role* and *situated commitment*. A role represents a coherent part of an agent's functionality in the context of an organization. We consider an organization as a group of agents that can play one or more roles and that work together. A situated commitment represents a social attitude of an agent, i.e. an agreement of an agent to play a particular role in an organization. Roles and situated commitments enable agents to set up collaborations and play different roles in such collaborations. Situated commitments in a collaboration typically depend on the actual context the involved agents are placed in. This approach enables agents to adapt their behavior with changing circumstances in the environment.

We have developed a model for *protocol-based communication* that enables situated agents to exchange messages according to communication protocols, i.e. well-defined sequences of messages. Message exchange is typically associated with cognitive agents, where the information encoded in the messages is related to mental state (beliefs, intentions, plans, etc.). This generally assumed perspective on communication does not fit the approach of situated multiagent systems. Protocol–based communication puts the focus of communication on the relationship between the exchanged messages. Direct communication enables situated agents to exchange information, and to set up collaborations reflected in mutual situated commitments.

1.5 Contributions

This research contributes with a new perspective on software engineering for multiagent systems. Especially, this research integrates situated multiagent systems as software architecture in a general software engineering process. Concrete contributions are:

- An advanced model for situated multiagent systems that links up with state-of-the-art approaches in the domain of multiagent systems. In particular, the model: (1) promotes the environment to a first-class abstraction that can be used creatively in the design of situated multiagent system applications [216, 223, 215]; and (2) extends state-of-the-art models for situated agents from mechanisms for action selection to an integral model that includes support for different concerns [206, 212], such as perception, social behavior, and direct communication.

- A set of mechanisms for architectural design, including: environment infrastructure for perception, action, and communication; laws that constrain the activity of agents; dynamics in the environment [230, 208, 209]; virtual environment [218]; selective perception [224, 228]; advanced action selection mechanisms with roles and situated commitments [226, 184]; protocol-based communication [227].

- A reference architecture that maps the advanced model for situated multiagent systems onto a system decomposition [210, 214, 213]. The reference architecture for situated multiagent systems integrates a set of architectural best practices from various applications we have studied and built. The reference architecture provides an asset base engineers can draw from when developing self-managing applications. The reference architecture also offers a vehicle to study and learn the advanced perspective on situated multiagent systems we have developed in our research.

  To demonstrate the feasibility of the reference architecture, we have developed an object-oriented framework that implements the architecture, and we have instantiated the framework for several prototype applications.

- The application of a situated multiagent system in a complex industrial application that uses automatic guided vehicles to transport loads through a warehouse [222, 221, 217, 177]. In particular, we have applied the various mechanisms for architectural design of situated multiagent systems to meet the functionality and satisfy the important quality attribute requirements of the system. The insights derived from the architectural design of this application have considerably contributed to the development of the reference architecture. The development of this application consisted of the following activities: (1) elicitation of the functional requirements of the system;

(2) elicitation and prioritization of the quality requirements; (3) incremental development of a software architecture; (4) evaluation of the software architecture; (5) incremental implementation of the application; (6) testing to verify the main system requirements.

- A disciplined evaluation of the software architecture of this industrial application [198, 211, 45]. For the elicitation and prioritization of the quality requirements and the evaluation of the software architecture we have applied the Architectural Tradeoff Analysis Method[1].

1.6 Overview

We conclude this introduction with an overview of this text. After the introduction, the text contains six chapters and a conclusion.

In chapter 2, we zoom in on architecture-centric software development, providing a detailed context in which situated multiagent systems will be used later on.

Chapter 3 explains our perspective on situated multiagent systems. We give an overview of the history of situated agent systems, and from this background we describe our perspective on situated multiagent systems. In this chapter, we introduce the Packet-World that we use as an illustrative case.

In chapter 4 we present the reference architecture for situated multiagent systems we have developed in our research. We describe different views of the reference architecture and we specify the variation mechanisms for applying the architecture to build concrete software architectures.

Chapter 5 elaborates on the development of an industrial transportation system as a situated multiagent system. Starting from the system requirements, we discuss the architectural design of the application, we explain how the software architecture of this application is related to the reference architecture, and how it has contributed to the development of the reference architecture. We zoom in on the evaluation of the software architecture and we discuss test results collected from an implemented system.

In chapter 6, we discuss related work that explicitly connects software architecture with multiagent systems. We also examine related work on the control of automated transportation systems.

Finally in chapter 7 we draw up conclusions. We summarize the contributions of this research and we report lessons we learned from applying multiagent system technology in practice. To conclude, we outline possible venues for future research.

[1] The Architectural Tradeoff Analysis Method (ATAM) is developed at the Software Engineering Institute [12] of Carnegie Mellon University. It is one of the most mature approaches for software architecture evaluation currently available [61].

Chapter 2

Architecture-Centric Software Development

Since the early 1990s, software architecture has been subject of increasing interest in software engineering research and practice. [61] gives three important reasons why architecture is important to software systems: (1) it is a vehicle for communication among stakeholders. Software architecture provides a basis for creating mutual understanding and consensus about the software system; (2) it is the manifestation of the earliest design decisions that have the most significant influence on system qualities. The tradeoffs between various qualities are all manifested in the software architecture; and (3) it is a reusable, transferable abstraction of a system. Software architecture constitutes a surveyable model for how a system is structured and works. This model is transferable to other systems with similar requirements and can promote large-scale reuse of design.

In this chapter, we zoom in on architecture-centric software development and provide a detailed context in which multiagent systems will be used later on. We start with situating architectural design in a software development life-cycle. Next, we briefly discuss requirement eliciting, the preparatory step to start architectural design. Subsequently we discuss architectural design. In this step, the various system requirements are achieved by architectural decisions based on architectural approaches such as architectural styles or a reference architecture. We explain how a software architecture is documented, and we zoom in on the evaluation of software architecture. To conclude, we briefly explain how software architecture serves as a basis for detailed design and implementation of the system.

2.1 Architectural Design in the Development Life Cycle

To understand the context of our approach for software development with multi-agent systems, we first situate architectural design in a software development life cycle. We use the evolutionary delivering life cycle [135, 35], see Fig. 2.1. This life

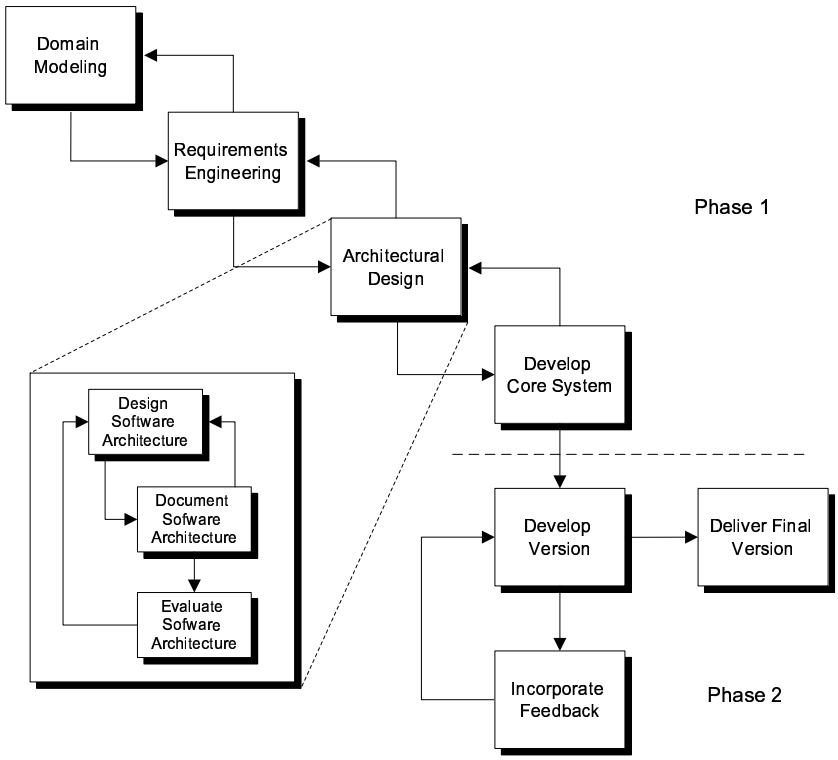


Figure 2.1: Architectural design in the software development life cycle

cycle model puts architectural design in the middle of the development activities. The main idea of the model is to support incremental software development and to incorporate early feedback from the stakeholders. The life-cycle consists of two main phases: developing the core system and delivering the final software product.

In the first phase the core system is developed. This phase includes four activities: defining a domain model, performing a system requirements analysis, designing the software architecture, and developing the core system. Requirements analysis includes the formulation of functional requirements of the system as well

as eliciting and prioritizing of the quality attributes requirements. Designing the software architecture includes the design and documentation of the software architecture, and an evaluation of the architecture. The development of the core system includes detailed design, implementation and testing. The software engineering process is an iterative process, the core system is developed incrementally, passing multiple times through the different stages of the development process. Fig. 2.1 shows how architectural design iterates with requirements analysis on the one hand, and with the development of the core system on the other hand. The output of the first phase is a domain model, a list of system requirements, a software architecture, and an implementation of the core of the software system.

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

2.2 Functional & Quality Attribute Requirements

Architectural design can start when the most important system requirements are known. This set of requirements are usually called the *architectural drivers* and include functional and quality attribute requirements.

Functionality is the ability of the system to perform the tasks for which it is intended. To perform a task, software elements have to be assigned correct responsibilities for coordinating with other elements to offer the required functionality. Functional requirements of a system are typically expressed as use cases, see e.g. [120]. A use case lists the steps, necessary to accomplish a functional goal for an actor that uses the system. In our research, we also use scenarios that describe interactions among parts in the system—rather than interactions that are initiated by an external actor. An example is a scenario that describes the requirement of collision avoidance of automatic vehicles on crossroads. Functionality does not depend on the structure of the system. In principle, if functionality were the only requirement, the system could exist as a single monolithic module with no internal structure at all [35]. Since functionality is largely independent of the structure of the software system, we do not go into details how functional requirements are collected.

Quality is the degree to which a system meets the nonfunctional requirements in the context of the required functionality. Quality attributes are nonfunctional properties of a software system such as performance, usability, and modifiability. Achieving quality attributes must be considered throughout the development process of a software system. However, software architecture is critical to the realization of most quality attributes, it provides the basis for achieving quality. For the expression of quality requirements we use *quality attribute scenarios* [34]. A quality attribute scenario consists of three parts:

1. Stimulus: an internally or externally generated condition that affects (a part of) the system and that needs to be considered when it arrives at the system; e.g. a user invokes a function, a maintainer makes a change, a component fails, etc.

2. Environment: the conditions under which the stimulus occurs; e.g. at runtime with system in normal operation, at design time, etc.

3. Response: the activity that is undertaken—through the architecture—when the stimulus arrives, the response should be measurable so that the requirement can be tested; e.g. the change requires a person-month of work, the error is displayed within five seconds, the system switches to save mode, etc.

Here is an example of a quality attribute scenario:

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

The stimulus in this example is "An Automatic Guided Vehicle (AGV) gets broken and blocks a path", the environment is "normal system operation", and the response is "other AGVs have to record this, choose an alternative route—if available—and continue their work".

Quality attribute scenarios provide a means to transform vaguely formulated qualities such as "the system shall be modifiable" or "the system shall exhibit acceptable flexibility" into concrete expressions. Scenarios are useful in understanding a system's qualities; formulating scenarios help stakeholders to express their preferences about the system in a clear way. Scenarios help the architect to make directed decisions and are a primary vehicle for analysis and evaluation of the software architecture. Ideally, the quality attribute scenarios of the system are collected and prioritized before the start of architectural design, e.g. during a quality attribute workshop [33]. Often however, a number of iterations will be necessary to gather and order system's requirements. In our research, we have used *utility trees* [61] to elicit and prioritize quality attribute scenarios. A utility tree provides a mechanism for the architect and the other stakeholders involved in a system to define and prioritize the relevant quality requirements precisely. We elaborate on utility trees in section 2.3.4 when we discuss the evaluation of software architecture.

2.3 Architectural Design

Designing a software architecture is moving from system requirements to architectural decisions. Besides a well-founded design method, this crucial engineering step requires thorough knowledge and experiences from architects.

In this section, we explain architectural design with a reference architecture. We explain the different types of views we have used to document a reference architecture, and we explain variation mechanisms to apply a reference architecture to design a concrete software architecture. After that we explain the approach for architectural design we have used in our research, and we motivate the need for good documentation of a software architecture. We conclude with explaining the evaluation of a software architecture.

2.3.1 Architectural Approaches and Reference Architecture

The achievement of a system's quality attributes is based on design decisions. Such decisions are called *tactics*. A tactic is a widely used architectural approach that has proven to be useful to achieve a particular quality [35, 171]. For example, "rollback" is a tactic to recover from a failure aiming to increase a system availability, or "concurrency" is a tactic to manage resource access aiming to improve performance. Actually, to realize one or more tactics an architect typically chooses an appropriate *architectural style* [180]. [35] defines an architectural style as "a description of architectural elements and relation types together with a set of constraints on how they may be used." An architectural style[1] is a recurring architectural approach that exhibits particular quality attributes. Examples of common architectural styles are layers, pipe-and-filter, and blackboard. Reference architectures [164, 35] go one step further in reuse of best practices in architectural design. A *reference architecture* integrates a set of architectural patterns that have proven their value for a particular family of applications. The Rational Unified Process [118] defines a reference architecture as "a predefined set of architectural patterns [...] proven for use in particular business and technical contexts, together with supporting artifacts to enable their use." A reference architecture serves as a blueprint for developing software architectures for a family of applications. Such family of applications is characterized by specific functionality and quality attribute requirements. Essential to a reference architecture are mechanisms to apply the reference architecture to a specific application. We use the term *variation mechanism* [85, 28, 142] to denote such mechanisms. The concept of a reference architecture is very similar to an application framework in object-oriented technology [104] and to a product line architecture [62]. In fact, the terms reference architecture and product line architecture are often used interchangeably [78].

Architectural Views. A reference architecture can be described by several *views* that emphasize different aspects of a software architecture. Building upon the work of Parnas [149] and Perry & Wolf [157], Kruchten introduced four main *views* of software architecture [117]. Each view emphasizes specific architectural aspects that are useful to different stakeholders. The logical view gives a description of

[1] An alternative term for architectural style often used in literature is "architectural pattern".

the services the system should offer to the end users; the process view captures the concurrency and synchronization aspects of the design; the physical view describes the mapping of the software onto the hardware and reflects its distribution aspects; and the development view describes the organization of the software and associates the software modules to development teams. A final additional view shows how the elements of the four views work together.

Based on [99], Rozanski and Woods [171] introduce the notion of a *viewpoint* that is defined as "a collection of patterns, templates, and conventions for constructing one type of view. It defines the stakeholders whose concerns are reflected in the viewpoint and the guidelines, principles, and template models for constructing its views." A viewpoint guides the architect in constructing a particular type of view. The authors put forward six core viewpoints: functional, information, concurrency, development, deployment, and operational. Orthogonal to viewpoints, the authors introduce the notion of architectural *perspective* to capture concerns that are common to many or all views. An architectural perspective is defined as "a collection of activities, tactics, and guidelines that are used to ensure that a system exhibits a particular set of related quality properties that require consideration across a number of the system's architectural views." Architectural perspectives guide the architect through the process of analyzing and modifying the software architecture to make sure it exhibits particular quality properties. Architectural perspectives include: accessibility, availability, evolution, internationalization, performance and salability, security and usability.

In [60], Clements and his colleagues generalize the notion of a view and relate views to architectural styles. The authors introduce the concept of a *viewtype* that defines the element types and relationship types used to describe the architecture of a software system from a particular perspective. Each viewtype constrains the set of elements and the relations that exist in its views. The authors distinguish between three viewtypes:

1. The module viewtype: views in the module viewtype document a system's principal units of implementation.

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

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

An architectural style is a specialization of a viewtype and reflects a recurring architectural approach that exhibit specific quality attributes, independent of any particular system. For example, "layered style" is a specialization of the module viewtype. The layered style describes (a part of) the system as a set of layers, each layer is allowed-to-use the services of the layer below. "Communicating processes

style" is an example of the component-and-connector viewtype. The communicating processes style describes concurrent units such as processes and threads, and the connection between the units such as synchronization and control. Finally, a view is an instance of an architectural style that is bound to specific elements and relations in a particular system.

In this dissertation, we follow this latter approach and use the notions of architectural style, view, and the three viewtypes—module, component-and-connector, and allocation viewtype—as presented by Clements et al. in [60].

Variation Mechanisms. In addition to views, a reference architecture specifies variation mechanisms and explains how these mechanisms can be applied to build a software architecture for a concrete system. A reference architecture can provide different variation mechanisms. Some examples are:

- Omission of an element: particular elements of the reference architecture are not common for every application; if applicable the architect can leave out such variable elements.
- Concretization of an abstract element: several elements in the reference architecture are abstractly defined and must be made concrete according to the requirements of the application at hand.
- Definition of behavior: the behavior of several elements is abstractly defined, the architect has to define concrete behaviors according to the requirements of the application at hand.

Some variation mechanisms are straightforward to apply, others however, may require additional support to help the architect. For some tasks established techniques are available, such as interaction diagrams and statecharts. However, other tasks require dedicated design guidelines. The documentation of a reference architecture should clearly describe what the applicable variation mechanisms are and how they can be exercised.

2.3.2 Architectural Design with a Reference Architecture

Architectural design requires a systematic approach to develop a software architecture that meets the required functionality and satisfies the quality requirements. In our research, we use techniques from the Attribute Driven Design (ADD [55, 35]) method to design the architecture for a software system with a reference architecture. ADD is a decomposition method that is based on understanding how to achieve quality goals through proven architectural approaches. Usually, the architect starts from the system as a whole and then iteratively refines the architectural elements, until the elements are sufficiently fine-grained to start detailed design

and implementation. At that point, the software architecture becomes a prescriptive plan for construction of the system that enables effective satisfaction of the systems functional and quality requirements [101, 60].

At each stage of the decomposition the architect selects an architectural element for refinement and determines the architectural drivers, i.e. the combination of functional goals and quality attribute scenarios that have to be realized. Then the architect selects an appropriate architectural approach that satisfies the drivers, decomposes the architectural element, and allocates functionality to the sub-elements.

A reference architecture can serve as a blueprint to *guide* the architect through the decomposition process. A reference architecture provides an integrated set of architectural patterns the architect can draw from to select suitable architectural solutions. The variation mechanisms specify how the abstract structures of the reference architecture can be applied to build a concrete design. However, it is important to notice that the reference architecture neither provides a ready-made cookbook for architectural design, nor should it stifle the creativity of the architect. Using a reference architecture does not relieve the architect from difficult architectural issues, including the selection of supplementary architectural approaches to deal with specific system requirements. A reference architecture 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.

In summary, for the architecture design of a software system with a reference architecture, the ADD process can be used to iteratively refine the software architecture, and the reference architecture can serve as a guidance in this decomposition process. In addition, common architectural approaches have to be applied to refine and extend architectural elements when necessary according to the requirements of the system at hand.

2.3.3 Documenting Software Architecture

To be effective, a software architecture must be well-organized and unambiguously communicated to the varied group of stakeholders. Therefore, good documentation of the software architecture is of utmost importance. The documentation must be general enough to be quickly understandable but also concrete enough to guide developers to construct the system. [60] gives three fundamental uses of architecture documentation:

1. Architecture serves as a means for education. Software architecture is a useful instrument to introduce new people to the system, such as new team members, external analysts, etc.

2. Communication among stakeholders. The software architecture represents a common abstraction that serves as a primary vehicle for communication

among stakeholders. Software architecture forms a basis for project organization, it imposes constraints on the design and implementation of the system, it is a starting point for maintenance activities, etc.

3. Software architecture serves as a basis for system analysis. The architecture must contain the necessary information to evaluate the various attributes; we elaborate on architecture evaluation in the next section.

The documentation of a software architecture consists of the relevant views completed with additional information that applies to different views. Views can be described with an architectural description language (ADL [136]). The ADL may be a formal or semi-formal descriptive language, a graphical language, or a combination of both. The Software Engineering Institute (SEI [12]) maintains a website [5] that provides links to websites of numerous ADLs. ADLs differ in many aspects, some languages are only intended for modeling, others offer support for formal analysis. Some ADLs are independent of any programming language, others are integrated with a particular programming language. This latter property is important to obtain guarantees that an implemented system conforms to its architecture. Many authors prefer the Unified Modeling Language (UML [13]) as ADL. Unfortunately, UML does not offer direct support for many architectural elements such as modules, layers and interface semantics. Since no ADL provides the facilities to completely document the various viewtypes we use to document software architectures (as explained in section 2.3.1), we employ a hybrid description language that uses UML constructs where possible. Each diagram is provided with a key that explains the meaning of the symbols used. Furthermore, we use a simple formal notation based on set theory to specify the various architectural elements and their relationships.

What views should be documented depends on the goals of the documentation. A software architecture intended for initial project planning likely contains another set of architectural views as an architecture that specifies the implementation units for development teams. Different views highlight different system elements and their relationships and expose different quality attributes. Therefore, the views that expose the most important quality attribute requirements of the stakeholders should be part of the architecture documentation.

Additional information of the software architecture documentation may include background information, a view template, a glossary, etc. For an example of a software architecture documentation, we refer to [43] that describes the various views and additional information of the software architecture of the automated transportation system we discuss in chapter 5.

2.3.4 Evaluating Software Architecture

A software architecture is the foundation of a software system, it represents a system's earliest set of design decisions [35]. These early decisions are the most

difficult to get correct, the hardest to change later, and they have the most far-reaching effects. Software architecture not only structures the system's software, it also structures the project in terms of team structure and work schedules. Due to its large impact on the development of the system, it is important to verify the architecture as soon as possible. Modifications in early stages of the design are cheap and easy to carry out. Deferring evaluation might require expensive changes or even result in a system of inferior quality.

Architectural evaluation is examining a software architecture to determine whether it satisfies system requirements, in particular the quality attribute requirements. Such evaluation focusses on the most important attributes, i.e. the attributes that are most important for the system's stakeholders and those that have the largest impact on the software architecture. Architectural evaluation typically takes place when the architecture has been specified, before implementation. This allows to add missing pieces, to correct inferior decisions, or to detail vaguely specified parts of the architecture, before the cost of such corrections would be too high.

The evaluation of software architecture is an active research topic, see e.g. [16, 145]. In our research we have used the Architecture Tradeoff Analysis Method (ATAM [61]) for evaluating a software architecture. ATAM is developed at the SEI, it is one of the most mature approaches for software architecture evaluation currently available. The general goal of ATAM is to determine the trade-offs and risks with respect to satisfying important quality attribute requirements. ATAM is an evaluation method that (1) uses stakeholders to determine important quality attribute requirements; (2) uses the architect to focus on important portions of the architecture; and (3) uses (or reveals the lack of) architectural approaches to determine potential problems. There are two groups of people involved in ATAM: the evaluation team and the stakeholders. The evaluation team conducts the evaluation and performs the analysis. The stakeholders are the people that have a particular interest in the software architecture under evaluation, such as the project manager, the architect, developers, costumers, (representatives of) end users, etc. An ATAM evaluation produces the following results:

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

point is an architectural decision that affects more than one attribute, it is a sensitivity point for more than one attribute.

A crucial document in the ATAM is the quality attribute utility tree, utility tree for short. This document is a prioritized list of quality attribute goals, formulated as scenarios. A utility tree expresses what the most important quality goals of the system are. An example of a utility tree is shown in Fig. 2.2.

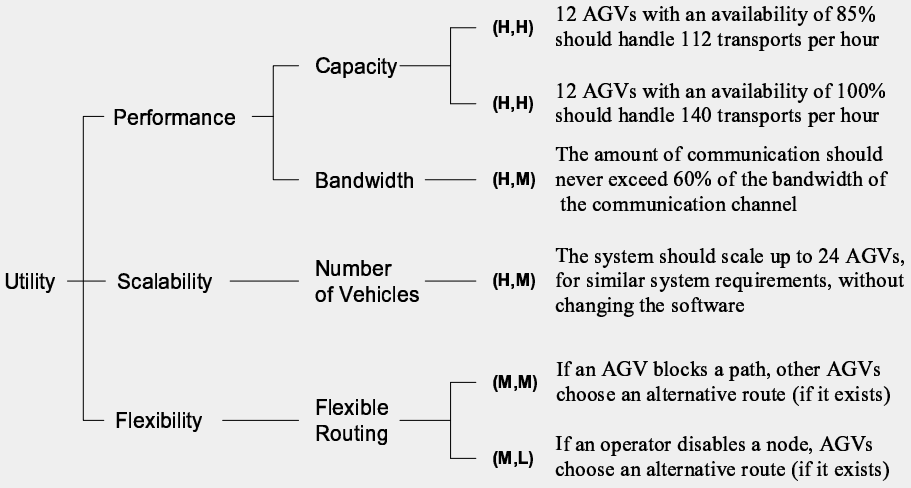


Figure 2.2: Sample utility tree

The root node of the tree is *utility*, expressing the overall quality of the system. High-level quality attributes form the second level of the tree. Each quality attribute is further refined in the third level. Finally, the leaf nodes of the tree are the quality attribute scenarios. Each scenario is assigned a ranking that expresses its priority relatively to the other scenarios, H stands for High, M for Medium, and L for Low. Prioritizing takes place in two dimensions. The first mark of each tuple refers to the importance of the scenario to the success of the system, the second mark refers to the difficulty to achieve the scenario. For example, the scenario "If an operator disables a node, AGVs choose an alternative route (if it exists)" has priorities (M,L), meaning that this scenario is of medium importance to the success of the system and relatively easy to achieve. The utility tree expresses what the most important qualities of the system are and as such it serves as a guidance for the evaluators to look for architectural approaches that satisfy the important scenarios of the system. It is clear that scenarios with priorities (H,H) and (H,M) are the prime candidates for analysis during the ATAM.

The evaluation of a software architecture with ATAM consists of three phases:

1. Presentations. The first phase consists of three steps: the evaluation leader starts by giving an overview of the evaluation method; next the project manager describes the business goals of the project; finally the architect gives an overview of the software architecture.

2. Investigation and analysis. The second phase also consists of three steps. First the architect identifies the architectural approaches applied in the software architecture. Next the quality attribute utility tree is generated. The system's quality attributes are elicited from the stakeholders and specified as scenarios. The list of scenarios is then prioritized. Finally, the architectural approaches that address the high-priority scenarios are analyzed, resulting in a list of risks, non-risks, sensitivity points, and tradeoff points. The analysis may uncover additional architectural approaches.

3. Reporting the results. In the final phase, the information collected during the ATAM is presented to the assembled stakeholders.

The flow of the ATAM is summarized in Fig. 2.3. The flow illustrates how the

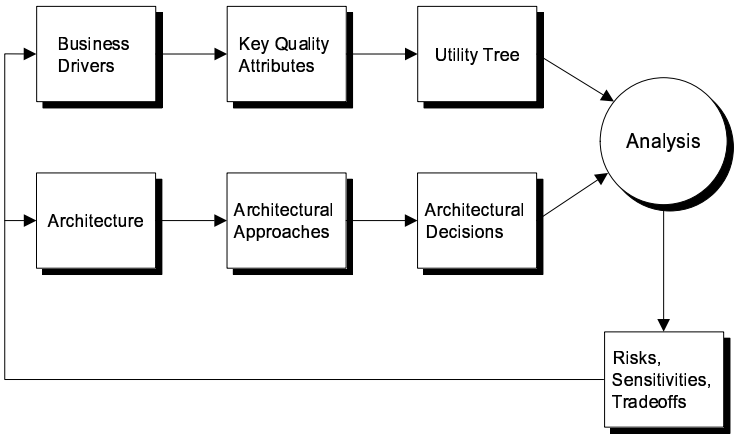


Figure 2.3: Conceptual flow of the ATAM

ATAM exposes architectural risks that may impact the software architecture and possibly the achievement of the organizations business goals.

2.4 From Software Architecture to Detailed Design and Implementation

A software architecture defines constraints on detailed design and implementation, it describes how the implementation must be divided into elements and how these elements must interact with one another to fulfil the system goals. On the other hand, a software architecture does not *define* an implementation, many fine-grained design decisions are left open by the architecture and must be completed by designers and developers. Examples are specific algorithms, internal data structures of modules, exception handling, etc.

2.5 Summary

In this chapter, we elaborated on architecture-centric software development and presented a detailed context in which multiagent systems will be used later in this dissertation. Starting from the system requirements, we discussed architectural design with particular attention for the role of a reference architecture. Rather than being a rigid frame that restricts the architect's creativity, the reference architecture is meant to serve as a valuable guidance for the architect that should be complemented with additional architectural approaches. We explained how a software architecture is documented, and we zoomed in on the evaluation of software architecture. By constraining detailed design and implementation of the system, software architecture provides the foundation for achieving systems requirements.

To conclude this chapter we give an overview how each of the following chapters is related to the presented development approach. Chapter 3 explains our perspective on situated multiagent systems, starting from an historical overview of situated agent systems. In chapter 4, we present a reference architecture for situated multiagent systems. Chapter 5 discusses the architectural design of a complex industrial transportation system with a situated multiagent system. We illustrate how we have used various mechanisms for adaptivity of situated agents for the architectural design of this application, and we zoom in on the ATAM evaluation of the software architecture. Finally in chapter 7 we reflect on our practical experience with the approach.

Chapter 3

Advanced Model for Situated Multiagent Systems

Around 1985, several researchers pointed to fundamental problems with deliberative agent systems [49, 14, 170]. Reasoning on internal symbolic models and action planning turned out to be insufficient for agents that have to operate in a dynamic and unpredictable environment. These researchers proposed radically new architectures for building agents. Whereas deliberative approaches emphasize explicit knowledge and rational choice, the emphasis of the new architectures is on direct coupling of perception to action, modularization of behavior, and dynamic interaction with the environment. This perspective—which is generally referred to as situated agent systems—was the start of a new vision on building agent systems. From these basic principles, many different approaches of situated agent systems have been developed and successfully applied in practice.

In this chapter, we explain the advanced model for situated multiagent systems we have developed in our research. First, we give an historical overview of situated agent systems, sketching the necessary background. Then we explain our model for situated multiagent systems, we show how this model links up with state-of-the-art approaches in the domain and contributes to the further development of the paradigm of situated multiagent systems.

3.1 Historical Overview of Situated Agent Systems

In this section, we make a tour through the evolution of situated agent systems. We start in the mid 1980s with the early single agent systems, and gradually evolve to stigmergic agents systems and situated multiagent systems. We reflect on the

architecture of subsequent agent systems[1] and we pay special attention to the role of the environment throughout the evolution.

3.1.1 Single Agent Systems

Initially, the research focus of situated agency was on single agent systems. In this section, we give an overview of the subsequent evolutions of single agent architectures and we discuss the role of the environment in this evolution.

3.1.1.1 Reactive Robotics

In the mid 1980s, researchers were faced with the problem of how to build autonomous robots that are able to generate robust behavior in the face of uncertain sensors, an unpredicted environment, and a changing world [51]. Attempts to build such robots with traditional techniques from artificial intelligence showed deficiencies such as brittleness, inflexibility, and no real-time reaction [122]. Besides, these systems suffered from several theoretical problems, such as the frame problem and the problem of non–monotonic reasoning within realistic time constraints [160]. This brought a number of researchers to the conclusion that reasoning on symbolic internal models, and planning the sequence of actions to achieve the goals is unfeasible for agents with many—often conflicting—goals that have to operate in complex, dynamic environments. This conclusion led to the development of a radically new approach to build autonomous agents. The key characteristics of this approach are described by Brooks in [51]:

- *Situatedness.* The robots are situated in the world, they do not deal with abstract descriptions, but with the here and now of the world directly influencing the behavior of the system.
- *Embodiment.* The robots have bodies and experience the world directly, they are in direct interaction with their environment.
- *Intelligence.* Robots are observed to be intelligent. The source of intelligence is not limited to the agents internal system, it also comes from physical coupling of the robot with the world.
- *Emergence.* The intelligence of the system emerges from the system's interactions with the world and from indirect interactions between its components.

Architectures for these robots emphasize a direct coupling of perception to action and the dynamic interaction with the environment. The environment is not only taken into account dynamically, but its characteristics are exploited to serve the

[1] In agent literature, the term "agent architecture" refers to the internal structure of an agent, in particular its decision making mechanism.

functioning of the system. The internal machinery of the robots typically consists of combinatorial circuits completed with timing circuitry. Each circuit represents a simple behavior of the agent. These circuits are hard-wired or pre-compiled from specifications. The resulting structure allows a robot to *react* in real-time to the changing conditions of the world in which it is embedded. Representative examples of approaches for reactive agents are Pengi [14] and Situated Automata [170]. In Pengi, the penguin's situated actions are coded in the form of simple rules. To formulate these rules, Pengi does not associate symbols with individual objects in the world, but uses expressions that describe causal relationships between the agent and entities in the world. These expressions use so-called indexical-functional representations of the environment. An example of a situated action is "if there is an ice-cube-besides-me then push ice-cube-besides-me". In Situated Automata, an agent is specified declaratively in the Gapps language [106]. From this specification a runtime program is generated, which satisfies the declarative specification. This program achieves real-time performance, it acts reactively without doing any symbol manipulation.

As an illustration of reactive robots, we briefly discuss the Subsumption Architecture developed by Brooks [49]. The subsumption architecture is organized as a series of parallel working layers, each layer is responsible for a specific behavior of the agent. The priority of layers—behaviors—increases from bottom to top. Higher layers are able to inhibit lower layers, giving priority to more important behavior. Fig. 3.1 depicts an example of a Subsumption Architecture for a simple robot that has to collect packets and deliver them at a destination. On its way, the robot must avoid obstacles in the environment.

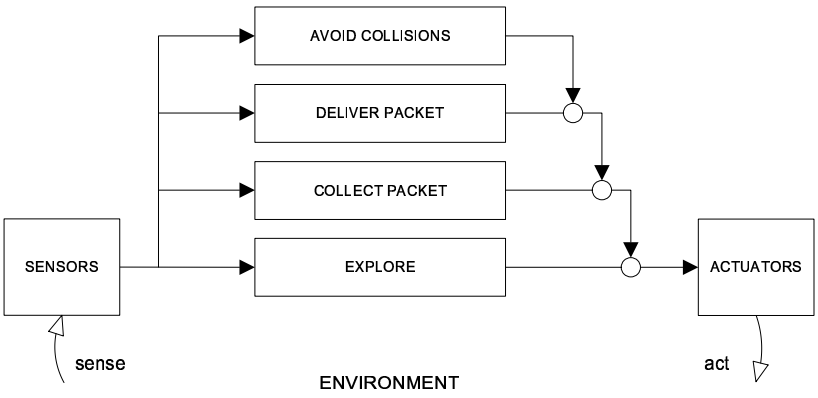


Figure 3.1: Subsumption Architecture for a simple robot

A layer in the architecture directly connects perception to action by means of a finite state machine augmented with timing elements. Each layer collects its own sensor data that is written in registers. The arrival of specific data, or the expiration of a timer, can trigger a change of state in the interior finite state machine and possibly produce output commands to actuators. Inhibition and suppression mechanisms resolve conflicts between actuator commands from different layers. In the original version of the subsumption architecture, finite state machines could not share any state [49]. Later this restriction was relaxed, allowing clusters of finite state machines (i.e. behaviors) to share state [50]. The Subsumption Architecture has successfully been used in many practical robots.

3.1.1.2 Behavior-Based Agents

In the early 1990s, researchers raised important limitations of the initial reactive approaches. In [122], Maes points to a number of problems with the wired or pre-compiled action selection structures of reactive architectures. Although these approaches demonstrate very good performance, they are typically very specific solutions, leaving little room for reuse. For complex agents in complex environments, the architectures are very hard to build. Another important shortcoming is the lack of explicit goals and goal-handling. The designer must anticipate what the best action is to take in all occurring situations. However, for complex systems much of the necessary information will only be available at runtime. Goals may vary over time and new goals may come into play.

Different approaches that support run-time decision making have been developed, usually referred to as behavior-based or situated agents. Prominent examples are Motor Schemas [19], Distributed Architecture for Mobile Navigation [168] (DAMN), and Free-Flow Architectures [169, 192]. The approach of motor schemas is based on schema theory that explains a robot's motor behavior in terms of the concurrent control of different activities [18]. A schema-based robot consists of a number of parallel executing motor schemas, each schema providing a behavior. Schemas can be added or removed at runtime. Each motor schema has as output an action vector that defines the way the robot should move in response to the perceived stimuli. The sum of output vectors determines the behavior of the robot. In DAMN different behaviors generate outputs as a collection of votes. Behavior arbitration is a winner-takes-all strategy in which the largest number of votes for an action is selected for execution. Multiple parallel arbiters for different control functions can be combined, e.g. for speed, turning, etc. A free-flow architecture consists of a tree of nodes which receive information from internal and external stimuli in the form of activity. The nodes feed their activity down through the tree until the activity arrives at the action nodes (i.e. the leaf nodes of the tree) where a winner-takes-all process decides which action is selected. A free-flow architecture allows an agent to take into account different preferences simultaneously.

As an illustration of behavior-based agents, we briefly discuss Maes' Agent Net-

work Architecture [122] (ANA). An ANA combines the robot-oriented principles of reactivity such as decomposition along tasks, de-emphasizing of internal world models, and emergent functionality, with goal-handling at runtime, and puts this approach in a broader context of software agent systems. An ANA is an action selection architecture of an agent. This architecture consists of a set of competence modules that are connected in a network, together with a mechanism to select a module for execution. Each competence module has its own specific competence that represents a particular behavior of the agent. A competence module has a list of preconditions which have to be true before the competence module becomes executable. In addition, each competence module has an activity level. When the activity level of an executable competence module reaches a certain threshold, it may be selected for execution, resulting in some actions. Fig. 3.2 shows a simple example of an agent network architecture.

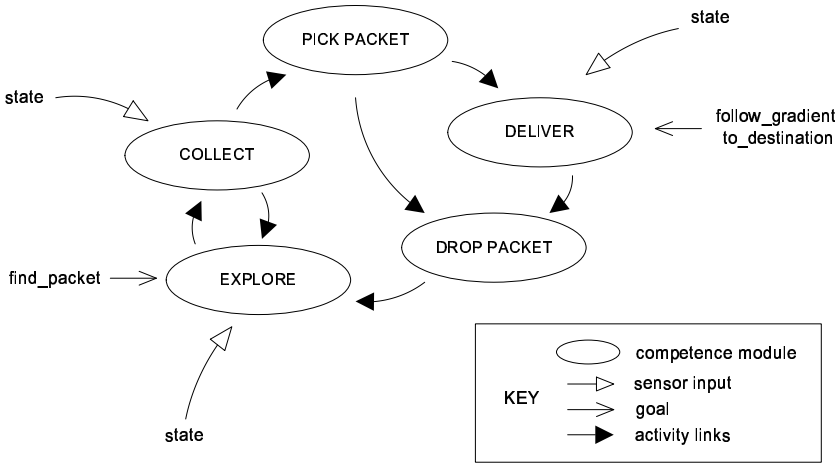


Figure 3.2: Agent Network Architecture for a simple robot

Competence modules are linked through different types of links. Modules use these links to activate and inhibit each other, so that after some time the activity level accumulates in the modules that represent the best actions to take, given the current situation and goals. The spreading of activity among modules as well as the input of new activity energy into the network is determined by the current observations and the goals of the agent. Note that goals may change at runtime. Through the cumulative effect of forward and backward spreading of activity along sequences of competence modules, the network exhibits implicit "planning" capabilities. The continuous re-evaluation of environmental input ensures that the action selection easily adapts with changing situations. However, Maes' ANA suffers also from a number of limitations, a detailed discussion is given by Tyrrell

in [192]. One problem is the loss of information because the approach assumes binary sensor data. However, many properties of realistic environments are continuous. Tyrrell has demonstrated that ANA suffers from an inherent unbalance of competition among competence modules, resulting in inefficient behavior. Another problem with ANA is the lack of compromise actions, i.e. ANA does not consider preferences of more than one competence module at a time. From our experiences [65], we learned that it is very difficult to design an agent network architecture for a non-trivial agent. ANA offers little support for structuring the behavior of complex agents. Moreover, adding a competence module to an existing network is almost impossible without affecting the existing structure.

3.1.1.3 Explicit World Models and Hybrid Agent Architectures

From the early start of situated agent systems, there has been an ongoing discussion about the exploitation of internal world models in agent architectures. Brooks argued against the need for any kind of world model or cognitive level at all [49]. Other researchers showed how knowledge may be compiled into non-symbolic implementations, see e.g. [105]. In [187], Steels states that "autonomous agents without internal models will always be severely limited". He proposes to use analogical instead of symbolic representations. A symbolic representation of the world is a description extracted from the perceived world in some language, e.g., a set of sentences expressed in first-order logic. An analogical representation, on the other hand, is a model of the world; examples are a map, a picture, and a diagram. Steels demonstrates the efficiency of analogical representations for a simple robot that has to acquire a map of the environment by wandering around. The agent emits a search wave, which places a diffuser field on a target when it finds it. Another argumentation for the necessity of knowledge representation was given by Arkin in [20]. Arkin states that "despite the assumptions of early work in reactive control, representational knowledge *is important* for robot navigation", and he demonstrates how a priori and dynamically acquired world knowledge can be exploited to increase flexibility and efficiency of reactive navigation. Related to the issue of explicit world models is the position of plans. In [15], Agre and Chapman elaborate on the use of plans in agents' decision making. The authors contrast two views on plans: plans as a resource to the agent versus plans for actions. In the view of plans as a resource, agents use plans as a resource among others in continually re-deciding what to do. In the view of plans for action, agents execute plans to achieve goals, i.e. a plan is a prescription of subsequent actions to achieve a goal. The analysis of Agre and Chapman laid the foundation for the work on reactive planning [141, 54].

In [125], Malcolm and Smithers introduced the notion of hybrid architecture. A hybrid architecture combines a deliberative subsystem with a behavior-based subsystem. The deliberative subsystem permits representational knowledge to be used for planning purposes in advance of execution, while the behavior-based sub-

system maintains the responsiveness, robustness, and flexibility of purely reactive systems. Over the years, many hybrid behavior-based architectures have been developed. Today, the approach is common in the domain of robotics, for an overview see [21]. A key element in hybrid architectures is the interface between reactivity and deliberation since it links rapid reaction and long–range planning. A common approach to balance reactivity with planning is to introduce an explicit third layer that coordinates among the reactive and deliberative layer. In general however, coordination of reactivity and deliberation is not yet well understood and is subject of active research.

3.1.1.4 Reflection

Starting from the initial principles of reactivity, a wide range of agent architectures have been developed. Three classes of approaches are identified:

1. *Reactive robots* emphasize the dynamic interaction with the environment. The internal machinery of the robots directly couples perception to action, enabling real-time reaction.

2. *Behavior-based agents* stress the need for dynamic and flexible action selection, aiming to cope with complex environments. Architectures for behavior-based agents support runtime arbitration among parallel executing behaviors and allow goals to vary dynamically over time.

3. *Hybrid agents* exploit representational knowledge about the environment. Architectures for hybrid agents integrate cognition (reasoning over internal representations of the world and planning) with reactivity (real-time reaction to stimuli) aiming to combine the advantages of planning and quick responsiveness.

These approaches share two properties:

1. The focus is on the *architecture* of *single* agents. Architectures differ in the way they solve the problem of *action selection*. Architectures do not support social interaction.

2. The approaches stress the importance of *environmental dynamics*. However, the environment itself is considered as the "external world", i.e. the environment is not a part of the models or architectures of the agent system itself.

3.1.2 From Collective Reactive Behavior to Situated Multiagent Systems

Since the early 1990s, researchers of situated agency have been investigating systems in which multiple agents work together to realize the system's functionality.

In these systems, the agents *exploit* the environment to share information and coordinate their behavior. In this section, we take a look at a number of relevant approaches that have been developed.

3.1.2.1 Collective Reactive Behavior

In [165], Reynolds demonstrated flocking behavior between a set of agents. The aggregate behavior of the multiagent system emerged from the interaction of multiple agents that each follows a set of simple behavioral rules. Mataric adopted these techniques to real robots [133], showing how a set of robots produced pack behavior. Each robot was provided with a set of simple behaviors from which it selects the most suitable behavior according to its current environmental context, i.e. its current position relative to other robots. In [243], Zeghal demonstrated another form of reactive coordination. Zeghal used vector fields to control the landing and movements of a large group of aircrafts in a simulation. In this approach, each agent is guided by a potential field that it constructs based on attracting and repulsing forces resulting from goals and obstacles (including other agents) respectively. An advanced example of behavior-based coordination among unmanned guided vehicles is demonstrated in the DARPA UGV programme[2]. In this case, a DAMN arbiter was used to coordinate the vehicle's behavior given its position in the formation. Although very attractive, several researchers have pointed to the complexity of designing collective reactive behavior, see e.g. [196, 21].

3.1.2.2 Stigmergic Agent Systems

In [83], Grassé introduced the term *stigmergy* to explain nest construction in termite colonies. The concept indicates that individual entities interact indirectly through a shared environment: one individual modifies the environment and others respond to the modification, and modify it in turn. Deneubourg [69] and Steels [186] demonstrated how explorer robots can improve the search of target objects by putting marks in the environment. When a robot finds a source of target objects, it puts a trail of marks in the environment from the source of objects toward the robot base, while returning home with an object. This trail allows other exploring robots to find the source of objects efficiently, similar to ants that inform each other about sources of food by depositing pheromone trails in the environment. To ensure that the robots are not mislead when the source becomes exhausted, the marks must be dynamic elements that vanish over time. This mechanism of indirect coordination through the environment combines reinforcement of the trail (positive feedback by the agent) with truth maintenance (decay of the trail over time by the environmnent). Stigmergy has been a source of inspiration for many multiagent system researchers. In [150], Parunak describes how principles of different natural agent systems (ants, wasps, wolves, etc.) can

[2]For a detailed discussion see [21].

be applied to build self-organizing artificial agent systems. Example applications of stigmergy are ant colony optimization [70], routing calls through telecommunication networks [41], supply chain systems [173], manufacturing control [52], and peer to peer systems [27].

We illustrate the use of marks in the environment with two prominent examples from literature: first we look at the Synthetic Ecosystem developed by Brueckner [52], after that we briefly discuss the Co-Fields approach proposed by Mamei and Zambonelli [128].

Synthetic Ecosystem. A synthetic ecosystem enables indirect coordination among software agents in the same way social ants coordinate; the software environment emulates the "services" provided by the real world of ants. The part of the software environment realizing the services is called the pheromone infrastructure. The pheromone infrastructure models a discrete spatial dimension. It comprises a finite set of places and a topological structure linking the places. A link connecting two places has a downstream and an upstream direction. Each agent in a synthetic ecosystem is mapped to a place, i.e. the current location of the agent, which may change over time. The pheromone infrastructure models a finite set of pheromone types. A pheromone type is a specification of a software object comprising a strength-slot (real number) and other data-slots. For each pheromone type, a propagation direction (downstream or upstream) is specified. The pheromone infrastructure handles a finite set of software pheromones for each pheromone type. Every data-slot is assigned a value of a finite domain to form one pheromone (type, direction, propagation, evaporation, etc.). The strength value (i.e. the value in the strength-slot) is interpreted as a specific amount of the pheromone. Different pheromones of a synthetic ecosystem may be stored in each place.

The pheromone infrastructure manipulates the values in the strength-slot of the pheromones at each place in three different ways:

1. External input (aggregation): Based on a request by an agent, the strength of the specified pheromone is changed by the specified value.

2. Internal propagation (diffusion): When an agent injects pheromone at a place, the input event is immediately propagated to the neighbors of that place in the direction of the pheromone. There the local strength of the pheromone is increased with the arriving pheromone value reduced by the propagation parameter. This process is recursively repeated until the remaining pheromone value crosses a minimal threshold.

3. Without taking changes caused by external input or propagation into account, the strength of each pheromone is constantly reduced in its absolute value (evaporation). The reduction is influenced by the evaporation parameter of the pheromone.

The pheromone infrastructure realizes an application-independent support for synthetic ecosystems designed according to a number of design principles, such as decentralization, locality, parallelism, indirect communication, information sharing, feedback, randomization, and forgetting. In [52, 150], Brueckner and Parunak describe a set of engineering principles for designing synthetic ecosystems, including: agents are things, not functions – keep agents small – decentralize control – support agent diversity – enable information sharing – support concurrency.

The principles of synthetic ecosystems and the proposed pheromone infrastructure are applied to a manufacturing control system [52]. Parunak and his colleagues have applied digital pheromones in many other practical applications, for an overview we refer to [9].

Co-Fields. Computational Fields (Co-Fields) is an approach to model and engineer the coordinated movements of a group of agents such as mobile devices (possibly carried by users), mobile robots, and sensors of a dynamic sensor network. In Co-Fields, the movements of the agents are driven by abstract (computational) force fields. By letting agents follow the shape of the fields, global coordination and self-organization can emerge.

The Co-Fields model is essentially based on the following three principles:

1. The environment is represented by fields that can be spread by agents or by the environment itself. These fields convey useful information for the agents to coordinate their behavior.

2. The coordination among agents is essentially realized by letting the agents follow the waveform of these fields.

3. Environment dynamics and movements of the agents induce changes in the surface of the fields, realizing a feedback cycle that influences agents' movement. This feedback cycle enables the system (agents and environment) to auto-organize.

A computational field is a distributed data structure characterized by a unique identifier, a location-dependent numeric value, and a propagation rule. Fields can be generated by the agents and by the environment, and are propagated through the space according to the propagation rule. The propagation rule determines how the field should distribute in the environment to shape of the field surface. Fields can be static or dynamic. A field is static if its magnitude does not change over time, while the magnitude of a dynamic field may change. Agents combine the values of the fields they perceive, the resulting new field is called the agents' coordination field. Agents follow (deterministically or probabilistically) the shape of their coordination field. Agents can follow the coordination field downhill, uphill, or along one of the equipotential lines of the field. Complex movements are achieved by dynamically re-shaping the surface of the field.

In principle, the approach can be generalized toward coordination fields spread in abstract spaces to encode coordination among agents that is related to actions differently from physical movements. In such a case, the agents follow their coordination field, not by moving from one place to another, but by making other kinds of actions.

The Co-Fields model is applied to a number of experimental applications, including a case study in urban traffic management [130] and a video game [129].

3.1.2.3 Situated Multiagent Systems

Stigmergic agent systems have proven their value in practice, yet, a number of comments are in order:

- Stigmergic agents are considered as "simple" entities. However, there is little or no attention for the architecture of agents.
- Stigmergic agents are not able to set up explicit collaborations to exploit contextual opportunities.
- The environment is considered as *infrastructure for coordination*, typically supporting one particular form of coordination. Such infrastructures provide reusable solutions that can be applied over many applications. Yet, choosing for a particular infrastructure compels an engineer to this approach and this may restrict flexibility.

Motivated by these considerations, researchers have extended the vision of stigmergic agents and developed architectures for a family of agent systems that is generally referred to as situated multiagent systems.

Multilayered Multi Agent Situated System. In the Multilayered Multi Agent Situated System [30, 31] (MMASS) agents and the environment are explicitly modelled. MMASS introduces the notion of agent type which defines agent state, perceptual capabilities and a behavior specification. Agent behavior can be specified with a behavior specification language [29] that defines a number of basic primitives, such as emit (starts the diffusion of a field), transport (defines the movement of the agent), and trigger (specifies state change when a particular condition is sensed in the environment). MMASS models the environment as a multi-layered structure, where each layer is represented as a connected graph of sites (a site is a node of the graph in a layer of the environment). Layers may represent abstractions of a physical environment, but can also represent logical aspects, e.g. the organizational structure of a company. Between the layers specific connections (interfaces) can be defined that are used to specify that information generated in one layer, may propagate into other layers. In MMASS, agents can (1) interact through a reaction with agents in adjacent sites (a reaction is a synchronous change of state of the involved agents), (2) emit fields that are diffused in

the environment, (3) perceive other agents, (4) update their state, and (5) move to adjacent sites. MMASS has been applied in various application areas, an example is an intelligent automotive system that is discussed in [32].

Influence–Reaction Model. In [76], Ferber and Müller propose a basic architecture for situated multiagent systems. This architecture builds upon earlier work of Genesereth and Nilson [80]. Ferber and Müller distinguish between tropistic and hysteric agents. Tropistic agents are essentially reactive agents without memory, whereas hysteric agents may have complex behaviors that use past experiences for decision making. Central to the model is the way actions are modelled. The action model distinguishes between influences and reactions to influences. Influences are produced by agents and are attempts to modify the course of events in the world. Reactions, which result in state changes, are produced by the environment by combining influences of all agents, given the state of the environment and the laws of the world. This clear distinction between the products of the agents' behavior and the reaction of the environment provides a way to handle simultaneous activity in the multiagent systems.

As an alternative to the centralized synchronization model of Ferber-Müller, we have introduced an action model based on regional synchronization [203, 207, 205]. With regional synchronization, only the influences of agents that can interact with one another are combined. Regional synchronization scales better, but the decentralized algorithm to determine the sets of interacting agents is quite complex.

In [74], Ferber uses the BRIC formalism (Block-like Representation of Interactive Components) to model situated multiagent systems based on the influence-reaction model. In BRIC, a multiagent system is modelled as a set of interconnected components that can exchange messages via links. BRIC components encapsulate their own behavior and can be composed hierarchically.

3.1.2.4 Reflection

In multiagent systems, multiple agents work together to realize the system's functionality. We identified three classes of systems in which the environment has a central role:

1. Agents with *collective reactive behavior* follow a set of simple behavioral rules. Each agent is driven by what it perceives in the environment. The aggregate behavior of the multiagent system emerges from the local behavior of agents.

2. In *stigmergic agent systems*, the environment serves as a medium for coordination. Stigmergic agents coordinate their behavior through the manipulation of marks in the environment. The environment is an active entity that maintains processes independent of the activity of the agents. Stigmergic coordination combines reinforcement of interesting information (positive

feedback by agents) with truth maintenance (decay of information over time by the environment).

3. *Situated multiagent systems* emphasize the importance of architecture for agents and the environment. Agent activity is decoupled from activity in the environment. Agents' actions are subject to laws that represent domain specific constraints.

Important characteristics of these multiagent systems are:

1. Agents and the environment are explicit parts of the system, each with its specific responsibilities.

2. System functionality emerges from the indirect interactions of agents through the environment.

Although situated multiagent systems have been applied with success in practice, many issues remain open for further research. We list a number of important challenges:

1. To extend architectures for situated agents from mechanisms for action selection to integral architectures that include support for different concerns of agents, such as perception and communication.

2. To endow situated agents with abilities for explicit social interaction. This enables situated agents to set up collaborations and play different roles in such collaborations.

3. To promote the environment to a first-class abstraction[3] that can be used creatively in the design of situated multiagent applications.

In addition to these specific demands, a major challenge is to develop a principled methodology for engineering situated multiagent systems.

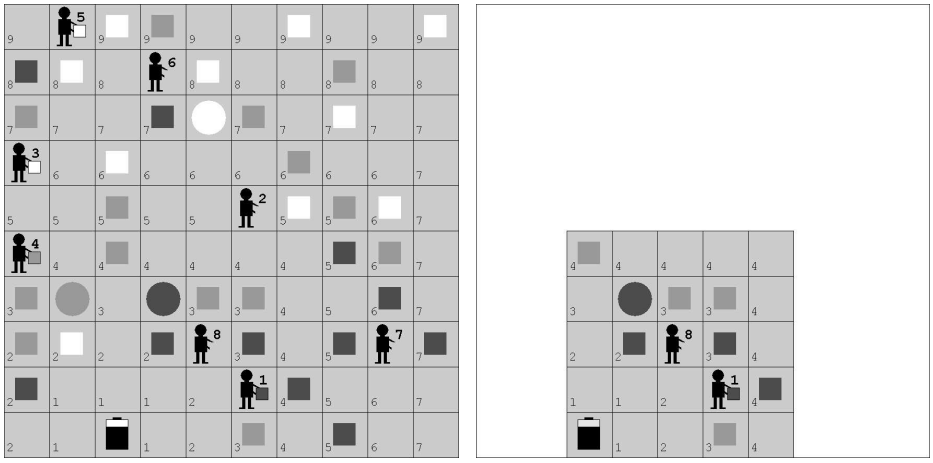
Our research links up with state-of-the-art approaches in the domain and contributes to the further development of the paradigm of situated multiagent systems by tackling these challenges. In particular, we integrate situated multiagent systems as software architecture in a general software engineering process. Central to this engineering approach is the reference architecture for situated multiagent systems. In the following sections, we present a model for situated multiagent systems that includes the basic functionalities to tackle the challenges mentioned above. This model provides the foundation of the reference architecture that we will discuss in the next chapter.

[3] [8] defines a first-class module as: "a program building block, an independent piece of software which [...] provides an abstraction or information hiding mechanism so that a modules implementation can be changed without requiring any change to other modules."

3.2 The Packet-World

Before we explain our perspective on situated multiagent systems, we first introduce the Packet-World that we will use as an illustrative case[4].

The basic setup of the Packet-World consists of a number of differently colored packets that are scattered over a rectangular grid. Agents that live in this virtual world have to collect these packets and bring them to the correspondingly colored destination. Fig. 3.3(a) shows an example of a Packet-World of size 10x10 with 8 agents (symbolized by the little fellows).



(a) A Packet-World of 10x10 (b) Local view of agent 8

Figure 3.3: Example of the Packet-World

Colored rectangles symbolize packets that can be manipulated by the agents and circles symbolize destinations. The battery symbol at the bottom row of the grid symbolizes a battery charger.

In the Packet-World, agents can interact with the environment in a number of ways. Agents can make a step to a free neighboring cell. A cell is free if it contains no object or agent (objects in the basic setup of the Packet-World are packet, destination, and battery charger). If an agent is not carrying any packet, it can pick up a packet from one of its neighboring cells. An agent can put down a packet it carries at one of the free neighboring cells, or of course at the destination point of that particular packet. Agents can also pass packets to neighboring agents

[4]The Packet-World is an experimental multiagent system application that is based on an idea of Huhns and Stephens [98]. We have developed and used the Packet-World quite extensively in our research as a test bed for investigating, experimenting and evaluating fundamental concepts and mechanisms of situated multiagent systems [200].

forming a chain. Such a chain enables agents to deliver packets more efficiently, e.g. in the situation of Fig. 3.3(a), agent 1 can pass a packet to agent 8 that can deliver the packet directly at the destination. Finally, if there is no sensible action for an agent to perform, it may wait for a while and do nothing. Besides acting in the environment, agents can also send messages to each other. In particular agents can request each other for information about packets or destinations and set up collaborations. The goal of the agents is to perform their job efficiently, i.e. with a minimum number of steps, packet manipulations, and message exchanges.

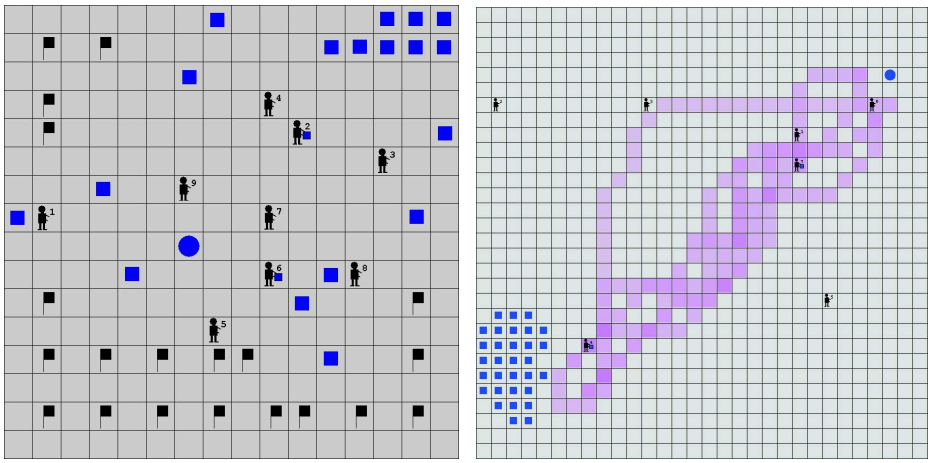
Agents in the Packet-World can access the environment only to a limited extent. Agents can only manipulate objects in their direct vicinity. The *sensing–range* of the world expresses how far, i.e. how many squares, an agent can perceive its neighborhood. Figure 3.3(b) illustrates the limited view of agent 8, in this example the sensing–range is 2. Similarly, the *communication–range* determines the scope within which agents can communicate with one another.

Performing actions requires energy. Therefore agents are equipped with a battery. The energy level of the battery is of vital importance to the agents. The battery can be charged at one of the available battery chargers. Each charger emits a gradient field, i.e. a force field that is spread in the environment and that can be sensed by the agents. The field values of all battery chargers are combined into a single field. To navigate towards a battery charger, the agents follow the gradient of the field in the direction of decreasing values. In the example of Fig. 3.3 there is only a single charger. The value of the gradient field is indicated by a small number in the bottom left corner of each cell. The intensity of the field increases further away from the charger.

In addition to the basic setup, the Packet-World also supports indirect coordination among agents via markers in the environment. Fig. 3.4 shows examples where agents coordinate via flags that demarcate areas where all packets have been collected, and digital pheromones that form paths between a cluster of packets and their destination (battery charging is omitted in these examples).

3.3 The Environment as a First-Class Design Abstraction

Now we put forward our perspective on situated multiagent systems and explain the advanced model for situated multiagent systems we have developed in our research [216, 230, 208, 209, 223, 215]. First, we direct our attention to the environment. We explain that engineers essentially consider the environment as infrastructure for agents and we motivate why this perspective does not exploit the full potential of the environment in multiagent systems. Then, we give a definition of the environment as a first-class design abstraction and we explain important functionalities that can be assigned to the environment.



(a) Flag-based coordination (b) Digital pheromone trails

Figure 3.4: Indirect coordination in the Packet-World

3.3.1 Levels of Support Provided by the Environment

From the historical overview presented in the first section of this chapter, we derive three levels of support provided by the environment that agents can use to achieve their goals.

Basic Level. At the basic level, the environment enables agents to access the *deployment context*. With deployment context, we refer to the given hardware and software and external resources with which the multiagent system interacts (sensors and actuators, a printer, a network, a database, a web service, etc.). Providing access to the deployment context to agents is an essential functionality of the environment in every agent system, it represents the most elementary perspective on the environment in an agent system. Fig. 3.5 depicts example scenarios in which agents directly access the deployment context.

In the example, the agent on the left side inserts two values into a table of a database. The agent in the middle opens a socket on a particular port number to contact another agent. The two agents on the right side access a shared printer. From these examples it becomes clear that direct access to the deployment context compels agents to deal with low-level details of the network, resources, and so on. Especially in dynamic and unpredictable deployment contexts such as ad hoc networks the agents' tasks become arduous.

Abstraction Level. The abstraction level bridges the conceptual gap between the agent abstraction and low-level details of the deployment context. The abstraction level provides an appropriate interface to the agents, shielding low-level details of

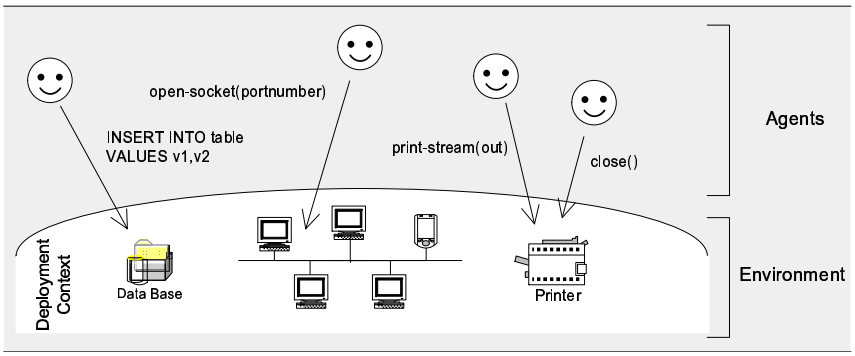


Figure 3.5: Agents directly access the deployment context

the network, legacy systems, and other resources external to the agent system. Fig. 3.6 depicts example scenarios of an environment containing an abstraction level.

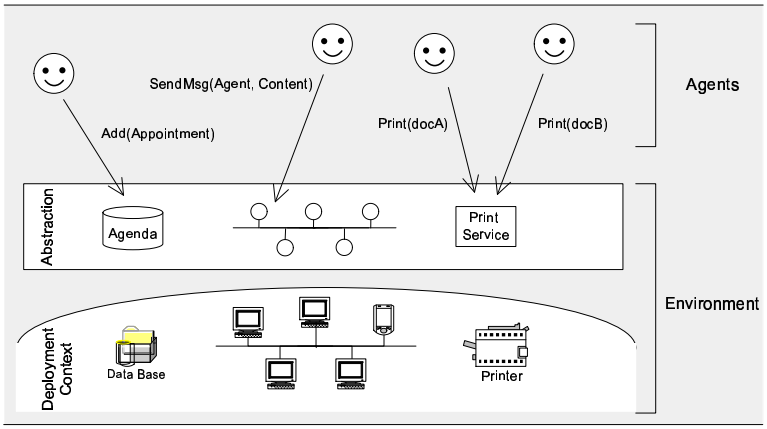


Figure 3.6: Abstraction level that shields agents from the details of the deployment context

Agents now access abstractions of the resources they are interested in. The Agenda repository allows agents to interact with the database at a higher level of abstraction. In the example, the agent adds an appointment in the Agenda. The abstraction level takes the burden of transforming the agents commands into SQL instructions to interact with the actual database. The network abstraction in the middle of Fig. 3.6 provides a communication infrastructure to agents to send and receive messages using agent names instead of sockets with ports and IP numbers,

etc. In a similar manner, the Print Service provides an abstraction of the printer that allows agents to instruct the service to print a document instead of sending streams of bytes to the output port, etc. Some other examples of functionality that can be supported by the abstraction level are mobility and fusion of sensor data.

In dynamic or unpredictable deployment contexts such as ad hoc networks, the abstraction level—typically supported by appropriate middleware—can shield the complexity of the deployment context (mobility, nodes that come and go, etc.) from the agents. An abstraction level of the environment is common in agent systems and is supported in most agent platforms.

Interaction-Mediation Level. In addition to the abstraction level, an environment can provide an interaction-mediation level to support mediated interaction in the environment. The interaction-mediation level offers support: (1) to regulate the access to shared resources, and (2) to mediate interaction between agents. Fig. 3.7 depicts example scenarios of interaction mediation.

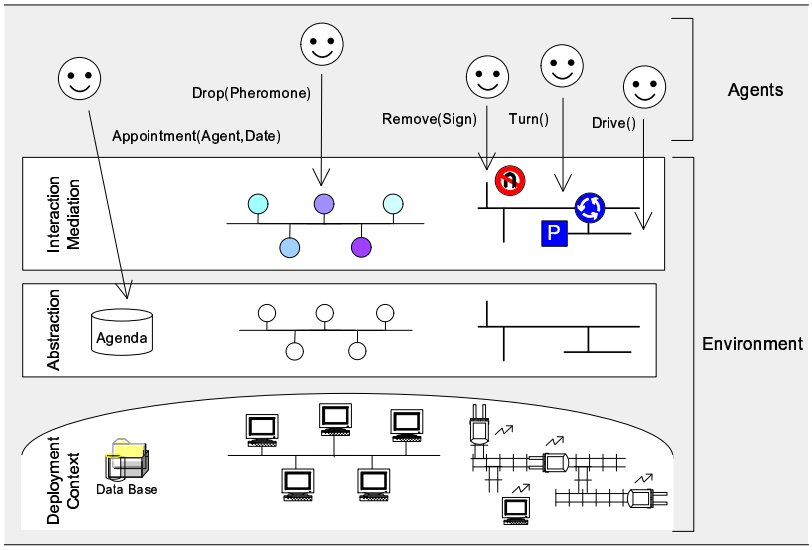


Figure 3.7: The environment mediates the interaction with resources and among agents

The agent in the middle of the figure interacts with a pheromone infrastructure that is deployed on top of a network topology. The agents on the right side steer automated vehicles. The abstraction level provides a map of the layout of the physical environment. The interaction mediation level adds signs to the map to regulate the traffic. Some of the signs may be manipulated by agents (e.g., the

agent on the right side removes the "no return" sign). Other signs may be managed by the environment; e.g., a sign that indicates the cost to drive to a particular destination may be adapted according to the actual traffic.

With an interaction-mediation level, the environment becomes an active entity in the system. The environment regulates particular activity in the system. Typically, the environment maintains activities independent of agent activities. Examples of such activities are aggregation, diffusion and evaporation of digital pheromones [52], maintenance of computational fields in a mobile network [131, 126], and support for tag-based coordination [158]. Support for interaction mediation enables agents to exploit the environment to coordinate their behavior.

Reflection. The three levels of support represent different degrees of functionality provided by the environment that agents can use to achieve their goals. In general, multiagent system engineers consider the environment essentially as *infrastructure* for coordination of agents. Such infrastructure provides a reusable solution that can be exploited over many applications. Yet, this perspective does not exploit the full potential of the environment in multiagent systems.

An infrastructure typically accounts for a specific set of responsibilities in the system. This hampers flexible assignment of responsibilities among agents and the environment. For a particular application, all responsibilities that are not managed by the infrastructure remain to be addressed by the agents, often leading to complex agents. Moreover, infrastructures are typically confined to a particular kind of coordination approach (marks, pheromones, fields, etc.). However, a solution may benefit from integrating different kinds of approaches according to the requirements of the system at hand. Today's multiagent system infrastructures are not developed with such kind of integration in mind. Finally, infrastructures typically focus on one set of responsibilities in the system. Communication infrastructures provide support for messages transfer, pheromone infrastructures provide support for a indirect coordination with digital pheromones, etc. The remaining functionalities of the environment often remain implicit or are dealt with in an ad-hoc manner, or even worse, agents are used to provide functionalities and services that conceptually do not belong to them.

In our research, we put forward the environment as a first-class design abstraction in multiagent systems, i.e. the environment is a building block that is considered explicitly and can be exploited creatively when building multiagent system applications.

3.3.2 Definition of the Environment as a First-Class Abstraction

Before we give our definition of environment, we first give a short overview of previous definitions of the environment described in literature.

Russell and Norvig [172] define a "generic environment program". This simple program gives the agents percepts and receives back their actions. Next, the program updates the state of the environment based on the actions of the agents and possibly other processes in the environment that are not considered as agents. Russell and Norvig's environment program illustrates the basic relationship between agents and their environment.

Rao and Georgeff [163] specify the following characteristics of the environment that are applicable to a broad class of agent system application domains: (1) at any instant in time there are potentially many different ways in which the environment can evolve; (2) at any instant in time there are potentially many different actions possible that agents can perform; (3) different objectives may not be simultaneously achievable; (4) the actions that best achieve the various objectives are dependent on the state of the environment; (5) the environment can only be sensed locally; (6) the rate at which computations and actions can be carried out is within reasonable bounds to the rate at which the environment evolves. Rao and Georgeff describe the typical characteristics of the external world in which agent systems are deployed and with which the agents interact.

Ferber [74] defines an environment as a space E, in which objects—including agents—are situated, i.e. at a given moment any object has a position in E. Objects are related to one another and agents are able to perceive objects and to manipulate (passive) objects in E. Agents' actions are subject to the "laws of the universe" that determine the effects of the actions in the environment. Ferber's definition underlines the container function of the environment and emphasizes the separation of agents' actions (as attempts to modify the course of events in the environment) and the reaction to the actions (i.e. the outcome of the actions) in the environment.

Demazeau [68] considers four essential building blocks for agent systems: agents (i.e., the processing entities), interactions (i.e., the elements for structuring internal interactions between entities), organizations (i.e., elements for structuring sets of entities within the multiagent system), and finally the environment that is defined as "the domain-dependent elements for structuring external interactions between entities." The environment in Demazeau's perspective emphasizes the structuring qualities of elements external to the agent system.

Parunak [150] defines an environment as a tuple $\langle State, Process \rangle$. *State* is a set of values that completely define the environment, including the agents and objects within the environment. *Process* indicates that the environment itself is an active entity. It has its own process that can change its state, independently of the actions of the embedded agents. The primary purpose of *Process* is to implement dynamism in the environment, such as maintenance processes of digital pheromones. Parunak's definition of environment underlines the active nature of the environment.

Odell et al. [144] define an environment as follows: "The environment provides

the conditions under which an entity (agent or object) exists". The authors distinguish between the physical environment and the communication environment. The physical environment provides the laws, rules, constraints and policies that govern and support the physical existence of agents and objects. The communication environment provides (1) the principles and processes that govern and support the exchange of ideas, knowledge and information, and (2) the functions and structures that are commonly employed to exchange communication, such as roles, groups, and the interaction protocols between roles and groups. Odell's definition of environment underlines the different structures of the environment (physical, communicative, social) and the mediating nature of the environment.

Our Definition. We define the environment as follows [215]:

> *The environment is a first-class abstraction that provides the surrounding conditions for agents to exist and that mediates both the interaction among agents and the access to resources.*

First of all, this definition states that the environment is *a first-class abstraction*. This stresses the fact that the environment is a building block in the multiagent systems that encapsulates its own clear-cut responsibilities, irrespective of the agents. Second, the environment *provides the surrounding conditions for agents to exist*. This implies that the environment is an essential part of a multiagent system. The environment is first of all the part of the world with which the agents interact, and in which the effects of the agents will be observed and evaluated. Furthermore, to build a useful system out of individual agents, agents must be able to interact. On their own, agents are just individual loci of control. The environment is the glue that connects agents into a working system. The environment encapsulates and provides access to external entities and resources, and enables agents to interact. Third, the environment *mediates* both the interaction among agents and the access to resources. This states that the environment can be an active entity with specific responsibilities in the multiagent system. The environment provides a medium for sharing information and mediating coordination among agents. As a mediator, the environment enables interaction and it constrains it.

Recognizing the environment as first-class abstraction promotes the environment to a design element that is considered explicitly and can be exploited creatively when building multiagent system applications. The environment can be assigned a custom set of responsibilities. Engineers can use agents as well as the environment to make a well-considered assignment of responsibilities according to the application requirements at hand. Allocating responsibilities among agents and the environment helps to manage the huge complexity of engineering real-world applications, and improves separation of concerns in multiagent systems.

3.3.3 Functionalities of the Environment

We now discuss a number of core functionalities that can be assigned to the environment. We illustrate the functionalities with examples of the Packet–World and other examples from literature. For an elaborated discussion, we refer to [209, 223, 215].

The Environment Structures the Multiagent System. The environment provides a shared "space" for the agents and resources, which structures the multiagent system. The agents and resources are dynamically related to each other. It is the role of the environment to define the rules which these relationships have to comply to. As such the environment acts as a *structuring* entity for the multiagent system. In general, different forms of structuring can be distinguished:

- *Physical structure* refers to spatial structure, topology, and possibly distribution, see e.g. [52, 30].
- *Communication structure* refers to infrastructure for message transfer or indirect communication, e.g., [52, 126].
- *Social structure* refers to the organizational structure of the environment in terms of roles, groups, societies, etc., see e.g. [75, 193, 176].

Structuring is a fundamental functionality of the environment. Structures of the environment may be imposed by constraints of the domain at hand, or they may be carefully considered design choices. In a distributed environment, physical and communication structure are part of the deployment context and should be supported by an appropriate level of abstraction. Social structure is typically supported at the interaction-mediation level.

The environment of the Packet-World is completely virtual. Agents in the Packet-World do not interact with resources external to the system. The physical structure of the Packet–World is a grid. The Packet–World provides an infrastructure for message transfer as well as infrastructure for indirect communication through markers in the environment such as fields, pheromones, and flags. An example of a social structure in the Packet–World are agents that collaborate in a chain to pass packets.

The Environment Embeds Resources. An important functionality of the environment is to embed resources. Resources are typically situated in a physical structure. The environment should provide support at the abstraction level shielding low-level details of resources and services to the agents.

Since agents in the Packet–World do not interact with external resources, the environment only includes virtual resources. Examples are packets, destinations, charger stations, and marks. Resources in the Packet–World are embedded in the grid structure of the environment. Some of the resources have a fixed position on

the grid, others have a position that can change over time.

The Environment Can Maintain Dynamics. Besides the activity of the agents, the environment can have processes on its own, independent of agents. An example of an environmental activity is a self-managing field in a network. When the topology of the physical network changes, the environment has to keep the fields consistent, see e.g. [126]. [175] introduces the notion of a *view* that offers an up-to-date application specific context representation to agents in a dynamic network. The environment may also provide support for maintaining state related to agents, such as tags that are used for coordination. Maintaining such dynamics is an important functionality of the environment. Depending on the nature of the dynamics, maintenance of dynamics can be supported at the abstraction level (e.g. maintenance of fused sensor data) or at the interaction-mediation level (e.g. maintenance of marks for coordination purposes).

In the Packet–World, the environment supports aggregation and evaporation of digital pheromones. When an agent drops new pheromone, the environment increases the strength of the local pheromone accordingly; and the environment decreases the strength of the pheromones over time.

The Environment is Locally Observable to Agents. Contrary to agents, the environment should be observable. Agents should be able to inspect the different structures of the environment, as well as resources, and possibly external state of other agents. Observation of a structure is typically limited to the current context (spatial context, communication context, and social context) in which the agent finds itself. In general, agents should be able to inspect the environment according to their current tasks. In our model for selective perception (see section 3.4.2), agents can use foci to direct their attention according to their current tasks. Agents should be able to observe the environment at the right level of abstraction. Observability of the deployment context should be supported at the abstraction level, support for observability of the social context is situated at the interaction-mediation level.

Agents in the Packet-World are able to observe the various resources on the grid in their vicinity. The identity of the agents is public in the Packet-World and as such also represented in the state of the environment.

The Environment is Locally Accessible to Agents. Agents must be able to access the different structures of the environment, as well as resources, and possibly external state of other agents. As for observability, accessing a structure is limited to the current context in which the agent finds itself. Access to spatial structure refers to support for metrics, mobility, etc. Access to communication infrastructure refers to support for direct communication (message transfer) and support for indirect communication (pheromones, etc.). Access to social structures

refers to organizations, group membership, etc. In general, resources can be perceived, modified, generated, and consumed by agents. The extent to which agents are able to access a particular resource may depend on several factors such as the nature of the resource, the capabilities of the agent, the actual relationships with other resources and agents, etc. Agents should be able to access the environment at an appropriate level of abstraction. Support for access of the social context is situated at the interaction-mediation level.

Agents in the Packet–World are able to access the resources in their vicinity. Agents can exchange messages with other agents that are located within communication range, and they can manipulate marks in the environment on neighboring cells.

The Environment Can Define Rules for the Multiagent System. The environment can define different types of rules on all entities in the multiagent system. Rules may refer to constraints imposed by the domain at hand (e.g. mobility in a network), or refer to laws imposed by the designer (e.g. limitation of access to neighboring nodes in a network for reasons of performance). Rules may restrict access to specific resources for particular types of agents, or determine the outcome of agents' interactions. Rules in the environment refer to shared access to resources and constraints on the interaction between agents. As such, support for rules is situated at the interaction-mediation level.

A simple example of a perception rule in the Packet-World is the sensing range that demarcates the scope agents can perceive their neighborhood. An example of a rule related to agents' actions is the restriction that an agent can only make a step to a free neighboring cell. Finally, a rule related to communication in the Packet-World is the rule that drops messages intended for agents outside the communication range of the sender.

3.4 Advanced Mechanisms of Adaptivity for Situated Agents

Now we zoom in on situated agents. Existing models of situated agents are typically restricted to mechanisms for action selection. Other concerns such as perception and communication are not dealt with, or integrated in the action selection model in an ad-hoc manner. The general motivation of the model for situated agents presented in this section is to provide better support for engineering situated agents. Particular goals are: (1) to provide explicit models for perception and communication and integrate these models with action selection; (2) to endow situated agents with abilities for explicit social interaction; (3) to provide mechanisms that enable agents to flexibly adapt their behavior with changing circumstances in the environment.

We start by explaining the state of agents. Then we discuss advanced mechanisms that cover different concerns of situated agents. Successively, we explain selective perception, advanced behavior-based action selection, and protocol-based communication.

3.4.1 Agent State

Contrary to knowledge-based agents, situated agents do not build up an internal model of the environment. Instead, they favor to employ the environment itself as a source of information. Situated agents use knowledge of the world to direct their decisions, however, this is done "here and now". A situated agent does not keep track of hypothetical future state or investigate the implications of state changes on a plan.

We distinguish between shared state and internal state. Both kinds of state can be further divided in static state and dynamic state.

- *Shared state* refers to state that is shared among agents. Examples are state that refers to elements observed in the environment, and state derived from data that is exchanged via messages.
  - *General static state.* This type of state refers to the agent's state of the system that does not change over time. Examples in the Packet-World are the grid structure of the environment, the set of possible colors of packets, and the identities of agents.
  - *Dynamic state.* This type of state relates to state about the agent's current context; it dynamically changes over time. Examples in the Packet–World are locally perceived packets and pheromones, and data about a temporal agreement for collaboration such as the identities of the agents with which the agent forms a chain to pass packets.
- *Internal state* refers to agent's state that is not shared with other agents. Internal state can be static, or it can dynamically change over time. Examples of internal static state are the various parameters of a behavior-based action selection mechanism. An example of internal state that dynamically changes is state that represents the success rate of recently selected behaviors. An agent can use such state to adapt its behavior over time, see e.g. [220].

3.4.2 Selective Perception

Perception is the ability of an agent to sense its neighborhood, resulting in a percept of the environment. A percept describes the sensed environment in the form of expressions that can be understood by the agent. Agents use percepts to update their knowledge of the world. Although perception is very common for any multiagent system, it is often dealt with in an implicit or ad-hoc manner. This is

especially the case for software environments where all aspects of perception must be modelled explicitly. The lack of attention for perception in multiagent systems was already raised in the mid nineties. In [123], Maes indicates the problem of the "narrow-minded view on the relationship between perception and action", pointing to the poor support for goal-driven perception. We have developed a model for *selective perception* in situated multiagent systems [224, 228]. Selective perception enables an agent to direct its perception at the relevant aspects of the environment according to its current task. This facilitates better situation awareness and helps to keep processing of perceived data under control.

To direct its perception an agent selects a set of *foci*. Each focus of the set of selected foci is characterized by a particular perceptibility, but may have other characteristics too, such as an operating range, a resolution, etc. Focus selection enables an agent to direct its perception, it allows the agent to sense the environment only for specific types of information. E.g., an agent `i` in the Packet–World that is interested in a "visible" perception of its neighborhood has to request a perception with `request(i, sense-objects())`, with `sense–objects()` the selected focus. As a result the agent will receive a representation of the elements within the default sensing range. A representation is a data structure that represents something in the environment. For the `sense–objects()` focus, the representation will consists of a data structure that represents the visible objects in the agent's sensing range. If the agent is only interested in the elements within a restricted range, it can add a parameter with its focus, e.g. `sense–objects(2)` which will yield a representation of the visible objects within a distance of two squares from the agent's position. On the other hand, an agent that is interested in the values of the fields that can guide it towards a battery charger has to select a focus `sense–fields()`. The representation will than consists of a data structure that contains the field values within the default sensing range.

To interpret a representation, agents use *descriptions*. A description is a blueprint that enables an agent to extract a percept from a representation. A percept consists of data elements that describe the sensed environment in the form that can be understood by the internal machinery of the agent. Consider for example a representation that contains a number of packets in a certain area. The agent that interprets this representation may use one description to interpret the distinguished packets and another description to interpret the group of packets as a cluster.

In addition to the selection of foci, selective perception enables an agent to select a set of *filters*. Filters allow the agent to select only those data elements of a percept that match specific selection criteria. Each filter imposes conditions on a percept that determine whether the data elements of the percept can pass the filter or not. E.g., an agent in the Packet–World that has selected a `sense–objects()` focus to visually perceive its environment and that is only interested in the destination of the yellow packets can select a filter `destination(packet(yellow))`.

The resulting percept will then contain a data element with the destination of the yellow packets (and only this destination)—at least, if this data element was part of the original percept, otherwise the resulting percept will be empty.

3.4.3 Behavior-Based Action Selection Extended with Roles and Situated Commitments

Decision making enables an agent to select appropriate actions to realize its tasks. We use a model for action that is based on Ferber's influence–reaction model, described in [76, 74]. We already touched on the influence–reaction model in section 3.1.2. In essence, this model separates what an agent wants to perform from what actually happens. Agents produce influences in the environment and subsequently the environment reacts to the influences resulting in a new state of the world. An example of an influence in the Packet–World is `influence(i, step(North))` which is produced by an agent with identity `i` that attempts to make a `step` on the grid in the direction `North`. Whether this influence will succeed depends on the agents capabilities (e.g., has the agent enough energy) as well as on the current situation in the environment (e.g., is the target cell currently accessible).

To select actions, a situated agent employs a behavior–based action selection mechanism. We have discussed a number of well-known behavior–based mechanisms for action selection in section 3.1.1. The main advantages of behavior–based action selection mechanisms are efficiency and flexibility to deal with dynamism in the environment.

3.4.3.1 Roles and Situated Commitments

As we explained in section 3.1, behavior–based action selection mechanisms are developed from the viewpoint of individual agents. Yet, in a situated multiagent system it is often desirable to endow agents with abilities for explicit social interaction. Explicit social interaction enables agents to exchange information directly with one another and set up collaborations. We extended behavior–based action selection mechanisms with the notions of *role* and *situated commitment* [225, 226, 185, 184, 227]. Roles and situated commitments provide the means for situated agents to set up collaborations.

We regard a role as a coherent part of an agent's functionality in the context of an organization. We consider an organization as a group of agents that can play one or more roles and that work together. Roles provide the building blocks for social organization of a multiagent system. This perspective on roles is similar to other approaches in agent research (see e.g. [107, 56, 143, 132]), provided that collaborations between situated agents are bounded to the locality in which the agents are situated.

Collaborations are explicitly communicated cooperations [153] reflected in mutual commitments. The attitude of a commitment has been studied extensively, however, always from the perspective of cognitive agents, see e.g. [63, 119, 71]. These traditional approaches take a psychological viewpoint on commitment, i.e. a commitment is essentially based on the mutual beliefs and the plans of the involved agents. We introduce the notion of a situated commitment as a social attitude of situated agents. Contrary to the traditional approaches on commitment which are essentially based on the mutually dependent mental states of the involved agents and a goal-oriented plan, a situated commitment is defined in terms of the *roles* of the involved agents and the local *context* they are placed in. Agreeing on a situated commitment incites a situated agent to give preference to the actions in the role of the commitment. We share the sociological viewpoint on commitment proposed in [182], however, that research focuses on cognitive agents in information–rich environments.

Agents agree on mutual situated commitments in a collaboration via direct communication (see section 3.4.4 below). Once the agents have agreed on a collaboration, the mutual situated commitments will affect the selection of actions in favor of the agents' roles in the collaboration. Whereas traditional approaches of commitment impose agents to communicate explicitly when the conditions for a committed cooperation no longer hold, for a situated commitment it is typically the local context in which the involved agents are placed that regulates the duration of the commitment. E.g., when agents form a chain in the Packet–World, the collaboration ends when no more packets are left to pass on, or when one of the agents leaves its post for maintenance. This approach fits the general principle of situatedness in situated multiagent systems. Note that an agent can also commit to itself. For example, if an agent in the Packet-World runs out of energy, it can commit to itself to resolve this urgent problem. Once committed, the agent will select actions in the role to recharge its battery: i.e. follow the gradient towards a battery charger, connect to the charger, and charge. The commitment ends when the battery is recharged.

3.4.3.2 Free-Flow Trees Extended with Roles and Situated Commitments

We now illustrate how we have extended free-flow trees, a concrete behavior-based action selection mechanism, with roles and situated commitments [226, 184]. We start by explaining how a traditional free-flow tree works. Then we show how we have integrated roles and situated commitments in free-flow trees.

Free-Flow Trees. Free-flow trees were first proposed by Rosenblatt and Payton in [169]. Tyrrell [192] has demonstrated that hierarchical free-flow architectures are superior to flat decision structures, especially in complex and dynamic environments. The results of Tyrrell's work are recognized in more recent research,

for a discussion see [54]. An example of a free-flow tree is shown in Fig. 3.8.

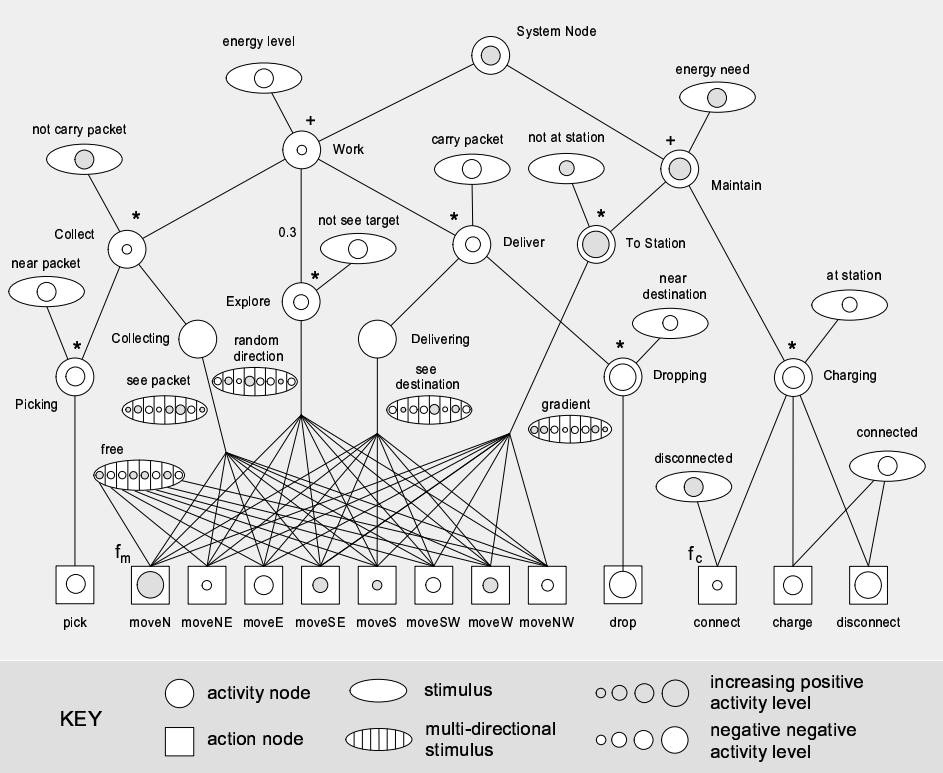


Figure 3.8: Free-flow tree for a Packet-World agent

A free-flow tree is composed of *activity nodes* (in short nodes) which receive information from internal and external stimuli in the form of *activity*. The nodes feed their activity down through the hierarchy until the activity arrives at the *action nodes* (i.e. the leaf nodes of the tree) where a winner-takes-all process decides which action is selected.

To explain decision making with a free-flow tree, we use the tree for a simple agent in the Packet-World, shown in Fig. 3.8. The left part of the tree represents the functionality for the agent to search, collect and deliver packets. On the right, functionality to maintain the battery is depicted. The *System Node* feeds its activity to the *Work* node and the *Maintain* node. The *Work* node combines the received activity with the activity from the *energy level* stimulus. The "+" symbol indicates that the received activity is summed up. The negative activity of the *energy level* stimulus indicates that little energy remains for the agent. As such

the resulting activity in the *Work* node is just below zero. The *Maintain* node on the other hand combines the activity of the *System Node* with the positive activity of the *energy need* stimulus, resulting in a strong positive activity. This activity is passed to the *ToStation* and the *Charging* nodes. The *ToStation* node combines the received activity with the activity level of the *not at station* stimulus (the "$\star$" symbol indicates they are multiplied). In a similar way the *Charging* node combines the received activity with the activity level of the *at station* stimulus. This latter is a binary stimulus, i.e. when the agent is at the charge station its value is positive (true), otherwise it is negative (false). The *ToStation* node feeds its positive activity towards the action nodes it is connected with. Each moving direction receives an amount of activity proportional to the value of the *gradient* stimulus for that particular direction. *gradient* is a multi-directional stimulus. The value of this stimulus (for each direction) is based on the sensed value of the gradient field that is transmitted by the charge station. In a similar way, the *Charging* node and the child nodes of the *Work* node (*Explore*, *Collect* and *Deliver*) feed their activity further downwards in the tree to the action nodes. Action nodes that receive activity from different nodes combine that activity according to a specific function. The action nodes for the move actions use a function f_m to calculate the final activity level. A possibility definition of this function is as follows:

$$A_{moveD} = \max\left[(A_{Node} \ + \ A_{stimulusD}) \ \star \ A_{freeD}\right]$$

Herein is A_{moveD} the activity collected by a move action, D denotes one of the eight possible directions, i.e. $D \in \{N, NE, E, SE, S, SW, W, NW\}$. A_{Node} denotes the activity received from a node, the move actions are connected to four nodes: $Node \in \{Collecting, Explore, Delivering, ToStation\}$. With each node a particular *stimulus* is associated. $stimulus \in \{see\ packet, random\ direction, see\ destination, gradient\}$ are all multi-directional stimuli with a corresponding value for each moving direction. Finally, *free* is a multi-directional binary stimulus that indicates whether the way to a particular direction is free for the agent to move to or not.

When all action nodes have collected their activity the node with the highest activity level is selected for execution. In the example, the *ToStation* node is clearly dominant over the other nodes connected to actions nodes. Currently the North-East, East, South-West and North-West directions are blocked (see the *free* stimulus), leaving the agent four possibilities to move towards the charge station: via North, South-East, South, or via West. The values of the gradient field guide the agent to move northwards, see Fig. 3.8.

The main advantages of free-flow architectures are:

- Stimuli can be added to the relevant nodes avoiding the "sensory bottleneck" problem [192]. In a hierarchical decision structure, to make correct initial decisions, the top level has to process most of the sensory information

relevant to the lower layers. A free-flow architecture does not "shut down" parts of the decision structure when selecting an action.

- Decisions are made only at the level of the action nodes; as such all information available to the agent is taken into account to select actions.
- Since all information is processed in parallel the agent can take different preferences into consideration simultaneously. For example, consider an agent in the Packet-World that spots two candidate packets to be picked at about equal distance. A Packet-World agent also has to maintain its battery. If the agent is only able to take into account one preference at a time it will select one packet and move to it, or alternatively it will follow the gradient field towards a battery charger. With a free-flow tree the agent can move towards one packet *while* it moves in the direction of a charge station, i.e. if the agent needs to recharge its battery in the near future, it will move towards the packet that is nearest to the battery charger.

Extending Free-Flow Trees with Roles and Situated Commitments. Free-flow trees are developed from the viewpoint of individual agents. To enable agents to exhibit explicit social behavior, we have extended the free-flow architecture with the abstractions of a role and a situated commitment. A role represents a coherent part of functionality of an agent in the context of an organization. Agents are related to one another by the roles they play in the organization. A role can consist of a number of sub-roles, and sub-roles of sub-sub-roles etc.

A situated commitment defines a relationship between one role (the goal role) and a non-empty set of other roles (the source roles) of the agent. When a situated commitment is activated, the behavior of the agent tends to prefer the goal role of the commitment over the source role(s). Favoring the goal role results in more consistent behavior of the agent towards the commitment. In a collaboration agents commit relatively to one another, typically via communication. However, an agent can also commit to itself, e.g. when it has to fulfill a vital task. A situated commitment is represented in the free-flow tree by a connector that connects the source roles of the situated commitment with the goal role. When a situated commitment is activated, extra activity is injected in the goal role relative to the activity levels of the source roles. Fig. 3.9 shows a free-flow tree for an agent in the Packet-World extended with roles and situated commitments.

A role corresponds to a subtree in the hierarchy. In the example, the roles are demarcated by dashed lines. The root node of a subtree that represents a role is denoted as the *top node* of the role. A role is named as its top node. Basic roles are roles that are not further divided in sub-roles. For the Packet-World agents, three main roles are distinguished: *Individual*, *Chain*, and *Maintain*. In the role Individual, the agent performs work, independent of the other agents. The agent searches for packets and brings them to the destination. The Chain role is composed of two sub-roles: *Head* and *Tail* denoting the two roles of agents

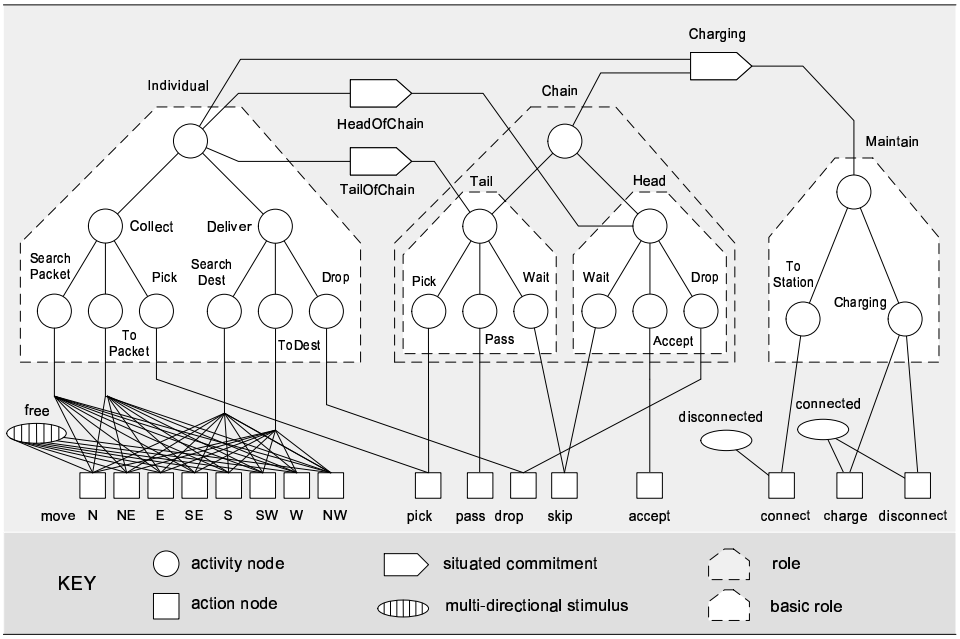


Figure 3.9: Free-flow tree for a Packet-World agent with roles and situated commitments (system node, combination functions, and stimuli of activity nodes omitted)

in a collaboration to pass packets along a chain[5]. Finally in the Maintain role, the agent recharges its battery. All roles of the agent are constantly active and contribute to the final decision making by feeding subsets of actions with activity. However, the contribution of each role depends on the activity it has accumulated from the affecting stimuli of its nodes.

A situated commitment is represented by a connector between roles in the tree. The connector *Charging* in Fig. 3.9 denotes the situated commitment of an agent to itself to recharge its battery. *Charging* connects the top nodes of the source roles *Individual* and *Chain* with the goal role *Maintain*. The connectors *HeadOfChain* and *TailOfChain* denote the mutual situated commitments of two agents that collaborate to pass packets in a chain. These situated commitments connect the single top node of *Individual* with the top node of *Head* and *Tail* respectively.

Fig. 3.10 shows the situated commitment *Charging* together with its goal role *Maintain* in detail. Besides a name, each situated commitment is characterized by an *activation condition*, a *deactivation condition*, and a 3-tuple ⟨*context, relation*

[5]To allow agents to set up a chain of more than two agents, an additional role *Link* would be necessary.

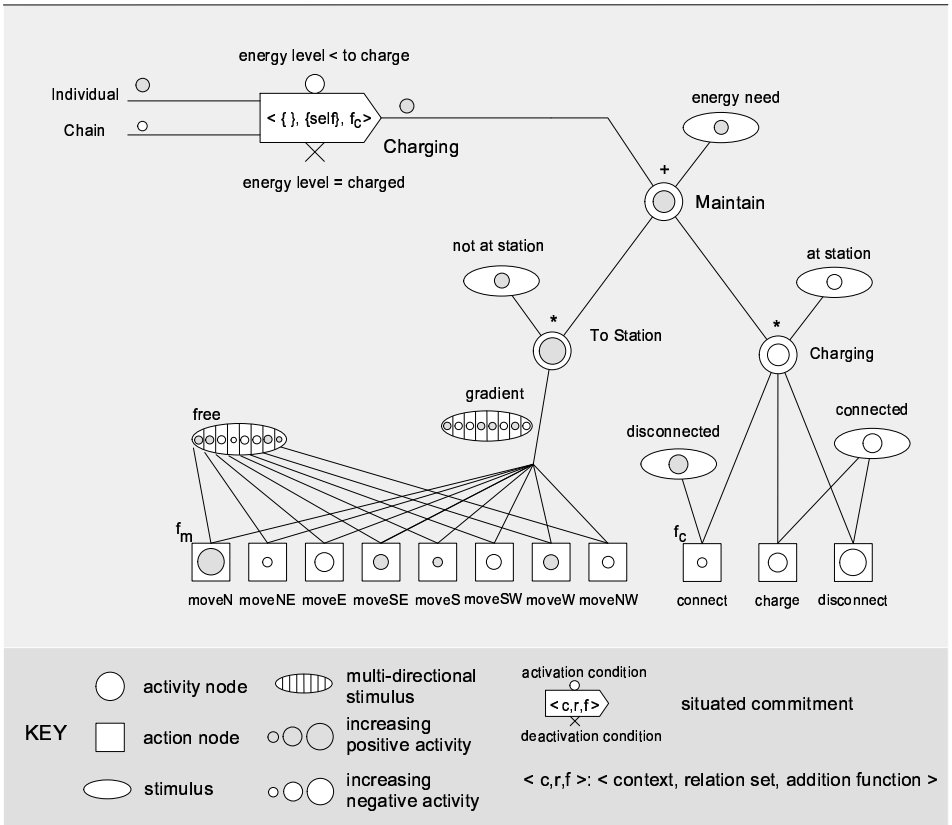


Figure 3.10: Situated commitment Charging with its goal role Maintain

set, addition function⟩. Activation and deactivation conditions are boolean expressions based on the agent's internal state. The activation condition for *Charging* in Fig. 3.10 is *energy level* $<$ *to charge*, i.e. as soon as the energy level of the agent crosses the threshold *to charge*, the activation condition becomes true and the situated commitment is activated.

The relation set contains the identities of the agents involved in the situated commitment. The context describes contextual properties of the situated commitment such as properties of elements in the environment (e.g., the color of the packets that are passed in a chain). Since *Charging* is a commitment of the agent relative to itself, the relation set is $\{self\}$. *Charging* is not related to any particular context. For example, for an agent that commits to be *HeadOfChain* in a collaboration (see Fig. 3.9), the relation set is the agent that is *TailOfChain*, and the context contains the kind of packets that are transferred between the collaborating

agents.

Finally, the addition function determines—once the commitment is activated—how the activities of the source roles are combined into a resulting activity that is injected in the goal role. When the *Charging* commitment is activated it injects an additional amount of activity in the *Maintain* role, determined by the addition function f_c. A possible definition for f_c is as follows:

$$A_{Charging} = A^{+}_{Individual} + A^{+}_{Chain}$$

$$\text{with } A^{+}_{Node} = A_{Node} \text{ iff } A_{Node} > 0, \text{ and 0 otherwise}$$

The *Maintain* role combines the additional activity of the *Charging* commitment with the regular activity accumulated from its stimuli. The deactivation condition of *Charging* is *energy level = charged*, i.e. as soon as the accumulated energy level reaches the *charged* level the commitment is deactivated. Then *Charging* no longer influences the activity level of its goal role.

In general, an agent can be involved in different situated commitments at the same time. The top node of one role may receive activity from different situated commitments and may pass activity to different other situated commitments. Activity received through different situated commitments is combined with the regular activity received from stimuli into one result.

3.4.4 Protocol-Based Communication

Communication enables agents to exchange information (e.g. agents in the PacketWorld can ask each other for the location of a packet or a destination), or to set up collaborations (e.g. agents can set up a chain to deliver packets more efficiently). Message exchange however, is typically associated with cognitive agents, where the information encoded in the messages is related to mental state (i.e., beliefs, plans, intentions, etc.). Yet, this generally assumed perspective on communication does not fit the approach of situated multiagent systems. We have developed a *protocol–based communication* model for situated agents that puts the focus of communication on the relationship between the exchanged messages. A communication protocol specifies a well-defined sequence of messages. We consider both binary and n-ary communication protocols. A binary protocol involves two communication peers (one as the initiator), whereas an n-ary protocol involves multiple communication peers (also with one initiator). Protocol-based communication is the interaction between agents based on the exchange of messages according to a specific communication protocol. We use the notion of *conversation* to refer to such an ongoing interaction. During the progress of a conversation agents typically modify their state implied by the communicative interaction. For example, when an agent agrees to start a collaboration, a situated commitment is activated which in turn will affect the agent's action selection according to its role in the

commitment. The end of a collaboration is typically induced by changes in the environment, but it may also be communicated explicitly by means of "end of cooperation" messages.

The information exchanged via a message is encoded according to a well defined communication language. A communication language defines the format of the messages, i.e. the subsequent fields the message is composed of. We denote the information decoded in a message *message data*. Message data describes the information of a message in a form that can be understood by the agent. The terminology of the modelled domain is defined in an ontology. The ontology defines: (1) a vocabulary of words that are used to represent concepts in the domain and (2) a set of relationships that express interconnections between these concepts.

3.4.4.1 Collaboration in a Chain in the Packet–World

As an example, we look at the communication protocol to set up a chain for passing packets in the Packet-World. A high–level description of the message sequence for this protocol is shown in the UML sequence diagram of Fig. 3.11.

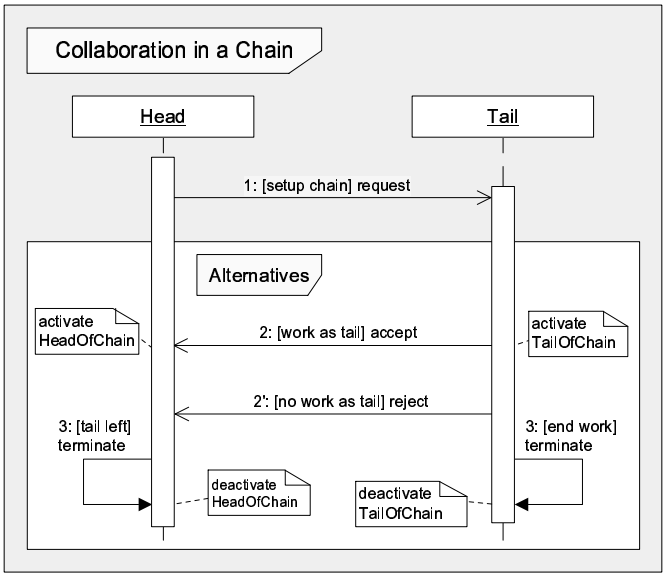


Figure 3.11: UML sequence diagram of the communication protocol to set up a chain in the Packet–World

If the conditions for an agent hold to enter the role of *Head* in a chain, it requests the candidate tail to cooperate. As an example, consider an agent with

identity i that requests an agent with identity j to form a chain for passing gray packets. The decoded message data of the received message may contain the following data:

$$\langle\ cid,\ j,\ request,\ \{Tail,\ chain(packet(gray))\}\ \rangle$$

This message data consists of four fields. *cid* is a unique identifier for the conversation. This identifier is assigned by the initiator of the conversation and is used by the participants to refer unambiguously to the conversation. j is the identity of the addressee, *request* is the performative of the message and $\{Tail,\ chain(packet(gray))\}$ is the content of the message. $chain(packet(gray))$ is an example of an expression that is defined in the agent's ontology. The expression connects three concepts of the agent's vocabulary *chain*, *packet* and *gray* into a relationship with an obvious semantic.

When the requested agent receives the request, it investigates the proposal. Depending on the context it answers with accept or reject. The conditions for the agent to accept the proposal are: (1) it knows the protocol to set up a chain; (2) the agent has the role *Tail*; (3) there are gray packets in the agent's neighborhood to pass on; and (4) the agent is not yet engaged in a commitment that conflicts with this new request. If all these conditions hold, the agent accepts the proposal, otherwise it rejects it. When accepting it, the agent becomes committed to the situated commitment *TailOfChain* and composes message data like:

$$\langle\ cid,\ i,\ accept,\ \{TailOfChain,\ packet(gray)\}\ \rangle$$

This message data contains the necessary information to encode the confirmation message of the collaboration.

After receiving the accept message, the initiating agent activates the situated commitment *HeadOfChain*. The cooperation is then settled and continues until all packets are passed or one of the agents in the chain for some reason leaves its position. When the collaboration ends, the situated commitments are deactivated.

In case the requested agent rejects the proposal, the conversation terminates without an agreement. Finally, when the requested agent neglects the request, e.g. when it urgently leaves to recharge its battery, the initiator detects this and terminates the conversation.

3.5 Summary

This chapter started with an historical overview of situated agent systems. From this background, we then explained an advanced model for situated multiagent systems that we have developed in our research.

Situated agency originates from reactive robotics that emerged in the mid 1980s as an approach to build autonomous robots that are able to act efficiently

in dynamic environments. The first generation of agent architectures directly coupled perception to action, enabling real-time reaction in the environment. Later, behavior-based architectures were developed that support runtime arbitration between parallel executing behaviors allowing the agents to act efficiently in more complex environments. The main focus of the early agent systems was on the architecture of agents, in particular on mechanisms for action selection. These architectures were developed for single agents and as such do not support social interaction. The environment itself is considered as the external world with which the agent interacts. With stigmergic agent systems, the environment became the center of attention. Stigmergic agents are driven by their surrounding environment. Internals of the agents are considered minimal. The environment serves as infrastructure for coordination. Recent research on situated multiagent systems places architecture for agents and the environment in the forefront. Agents and the environment are explicit parts of the system, each with its specific responsibilities.

Our research links up with state-of-the-art approaches in the domain and contributes to the further development of the paradigm of situated multiagent systems.

We explained our perspective of the role of the environment as a first–class abstraction in multiagent systems, we gave a definition of the environment, and we discussed important functionalities that can be assigned to the environment in multiagent systems. Functionalities include: (1) the environment structures the multiagent system; (2) the environment embeds resources; (3) the environment can maintain dynamics that happen independently of agents; (4) the environment is locally observable to agents; (5) the environment is locally accessible to agents; and (6) the environment can define rules for the multiagent system.

We presented an advanced model for situated agents that (1) integrates decision making with selective perception and protocol–based communication; (2) endows situated agents with abilities for explicit social interaction by means of roles and situated commitments.

The perspective on situated multiagent systems described in this chapter provides the foundation for the reference architecture we describe in the next chapter.

Chapter 4

A Reference Architecture for Situated Multiagent Systems

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. 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 chapter, we give an overview of the reference architecture for situated multiagent systems we have developed in our research [206, 228, 227, 214, 210, 213]. This reference architecture integrates the agent and environment functionalities we have discussed in the previous chapter and maps them onto an abstract system decomposition, i.e. software elements and relationships between them. This abstract system composition provides a blueprint for architectural design of the family of self-managing applications we target in this 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 in two parts. First, we give an integrated model for situated multiagent systems that synthesizes the agent and environment functionalities that are covered by the reference architecture. Then we present the reference architecture itself. The architecture documentation consists of four views that describe the reference architecture from different perspectives. To conclude, we refer to a framework that we have developed that demonstrates the feasibility of the reference architecture, and we end the chapter with a brief summary.

4.1 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.

4.1.1 Reference Architecture Rationale

In chapter 3, we presented our perspective on situated multiagent systems. We explained the role of the environment in multiagent systems and we discussed important functionalities that can be assigned to the environment. We also presented advanced mechanisms of adaptivity for situated agents, including selective perception, advanced behavior-based action-selection mechanisms with roles and situated commitments, and protocol-based communication. The various agent and environment functionalities provide a basis for developing self-managing situated multiagent systems. However, designing a software architecture for a practical application based on these functionalities is a complex matter.

The general goal of the reference architecture is to support the architectural design of self-managing applications with situated multiagent systems. Concrete motivations for the reference architecture are:

- *Integration of mechanisms.* The mechanisms of adaptivity for developing self-managing applications are described separately. However, 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.

4.1.2 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 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 operate in 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, automated transportation systems, and robotics.

The reference architecture abstracts from the concrete deployment of the multiagent system application, which highly depends on the particular system requirements. By abstracting from the distribution of the system functionality, the reference architecture is also valuable for non-distributed applications. In chapter 5, we discuss the design of a concrete software architecture for an automated transportation system and show how distribution is integrated with the functionality provided by the reference architecture for this industrial application.

4.1.3 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. [201, 203, 207, 59] investigate agents' actions in the Packet–World. [200, 137] study various forms of stigmergic coordination. [219, 220, 82] focus on the adaptation of agent behavior over time.

[202, 184, 92] yield 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 [224, 94]. This application applies a pheromone–based approach for the coordination of agents that move around in a dynamic network searching for files. [198, 44, 179, 178] study a field-based approach for task assignment to automatic guided vehicles that have to operate in a dynamic warehouse environment. Finally, [225, 226, 65, 92] study 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 [221, 222, 211, 7, 42].

Driven by the study and development of these applications, we incrementally developed various new mechanisms for situated multiagent systems as we presented in the previous chapter. In the course of building the various applications, we derived common functions and structures that provided architectural building blocks for the reference architecture. As such, the reference architecture integrates the different agent and environment functionalities we discussed in the previous chapter 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.

4.1.4 Organization of the Reference Architecture Documentation

The reference architecture documentation starts with an introductory part that describes an integrated model for situated multiagent systems. This model synthesizes the agent and environment functionalities that are covered by the reference architecture.

The second part of the documentation describes the various architectural views of the reference architecture. The documentation includes a module decomposition view and three component and connector views: shared data, collaborating components, and communicating processes. Each view is organized as a set of view packets [60]. 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 pre-

sentation 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. A detailed formal specification of the various architectural elements is available in Appendix A.

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.

4.2 Integrated Model for Situated Multiagent Systems

In this section, we present an integrated model for situated multiagent systems that synthesizes the environment and agent functionalities covered by the reference architecture. This model integrates the various mechanisms of adaptivity for situated multiagent systems we presented in chapter 3. For clarity, we have divided the model in two parts: environment and situated agent. We discuss both parts in turn.

4.2.1 Model of the Environment

The environment model consists of a set of modules with flows between the modules [215, 216, 223, 209, 230]. The modules represent the core functionalities of the environment. Fig. 4.1 shows the environment model.

The model consists of two main modules, the *deployment context* and the *application environment*. With the term deployment context, we refer to the given hardware and software and external resources with which the multiagent system interacts (sensors and actuators, a printer, a network, a database, a web service, etc.). With the term application environment, we refer to the part of the environment that has to be designed for an application, i.e. the functionality on top of the deployment context. As an illustration, the environment in the Packet-World only consists of the application environment. Agents in the Packet-World do not interact with resources external to the system. The deployment context as well as the modules of the application environment that interact with the deployment context are absent in the Packet-World. The application environment enables the

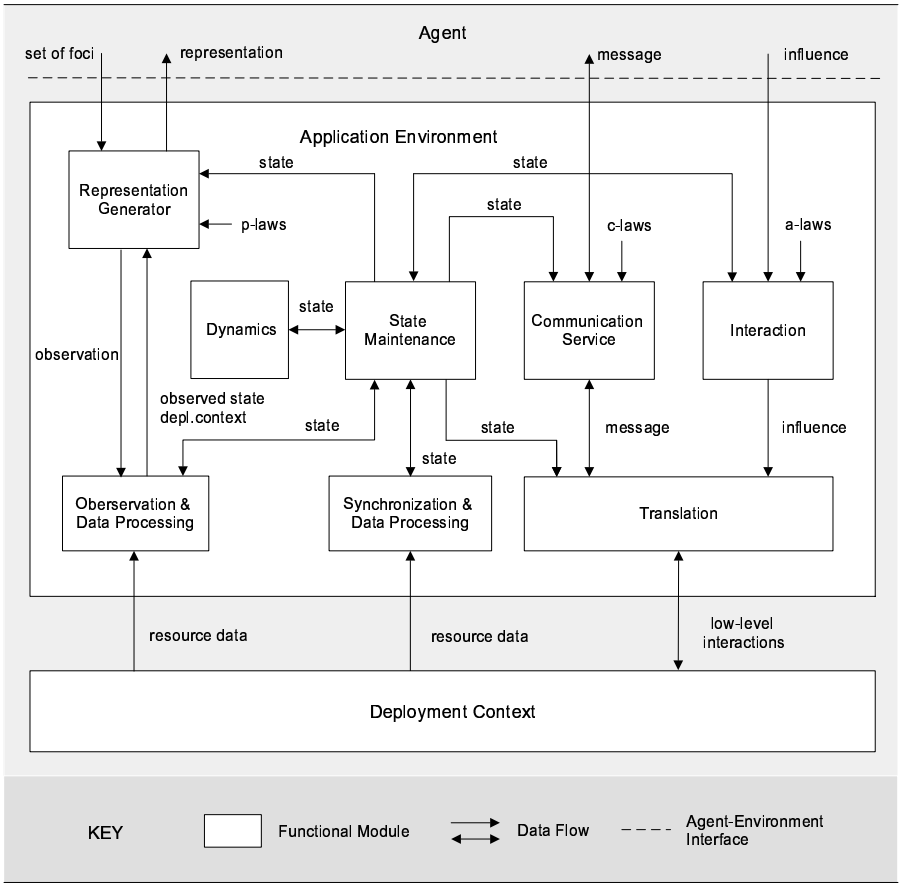


Figure 4.1: Environment model

agents to perceive objects in their vicinity, to pick up and drop off packets, to exchange messages, to move to free neighboring cells, etc. Another example is a situated multiagent system applied in a peer-to-peer file sharing system. Such application 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 on top of the deployment context, which enables agents to coordinate their behavior. E.g., the application environment can offer a pheromone infrastructure to agents that they can use to dynamically form paths to locations of interest.

Our focus is on the application environment, i.e. the part of the environment

that has to be designed for an application. The model of the application environment covers the different functionalities of the environment discussed in the chapter 3. The decomposition is considerably determined by the way agents interact with the environment. An agent can sense the environment with a *set of foci* to obtain a *representation* of its vicinity, an agent can invoke an *influence* in the environment attempting to modify the state of affairs in the environment, and it can exchange *messages* with other agents. Whereas influences are concerned with direct manipulation of the state of affairs in the environment, exchanging messages are not. Communicative interaction occurs in a sequential manner and concerns the coordination of actions among agents. Communicative interaction enables agents to resolve conflicts, to request each other for services, to establish a future cooperation, etc. Considering perception, action and communication as distinct ways to access the environment is shared by many multiagent system researchers, see e.g. [74, 144, 240]. We now explain the different modules of the application environment.

State Maintenance has a central role in the application environment; this module provides functionality to the other modules to access and update the state of the application environment. The state of the application environment typically includes an abstraction of the deployment context and possibly additional state. 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 a map of the physical environment. The environment state may also include agent-specific data, such as agents' identities and positions, and tags used for coordination purposes. The state of the application environment in the Packet-World includes a representation of the grid, packets, destinations, charger stations, fields, etc. The identity of the agents is public in the Packet-World and as such also represented in the state of the application environment.

Representation Generator provides the functionality to agents for perceiving the environment selectively. The main functions of the representation generator are *perception restriction*, *state collection*, and *representation generation*. To sense the environment, an agent invokes a perception request with a set of foci. Agents' perception is subject to perception laws (p-laws). Perception restriction applies the perception laws to the set of foci, restricting what the agent is able to perceive. For example, for reasons of efficiency a designer can introduce limits on agents' perception in order to restrain the amount of information that has to be processed. Perception laws can be defined relatively to the actual state of the application environment, enabling the perception generator to adapt the restrictions on perception according to changing circumstances. A simple example of a per-

ception law in the Packet-World is the sensing–range that demarcates the scope agents can perceive their neighborhood. State collection collects the observed state from the application environment, possibly completed with state derived from the observation of the deployment context. Representation generation converts the observed state into a representation that is returned to the requesting agent. An example of a representation in the Packet-World is a data structure that describes the types and values of the digital pheromones in an agent's vicinity.

Observation & Data Processing provides the required functionality to observe the deployment context. The main functions of observation & data processing are *data retrieving* and *data processing*. When agents request to sense resources in the deployment context, the representation generator uses the observation & data processing module to retrieve the requested data. Data obtained from the observation of resources in the deployment context is passed to the representation generator that produces representations for the requesting agents. Rather than delivering raw data retrieved from recourses in the deployment context, the observation module can provide additional functions to accommodate the sensors used in real-world applications. Data processing can include sorting of data, sensor calibration, data correction, data interpolation, etc.

Interaction deals with agents' actions in the environment. To model actions, we use the influence–reaction model introduced by Ferber [74]. This model distinguishes between influences, which are produced by the agents and which attempt to modify the course of events in the environment, and reactions which are produced by the environment and which result in state changes of the environment. The core functions of the interaction module are *collector*, *operation restriction*, and *reactor*. The collector collects the influences invoked by the agents. Agents' influences can be divided in two classes: influences that attempt to modify the state of the application environment, and influences that attempt to modify the state 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 (a-laws). Action laws put restrictions on the influences invoked by the agents. Operation restriction applies the set of action laws to the agents' influences. Simple examples of action laws in the Packet-World are a law that imposes the restriction that an agent can only pick a packet from a neighboring cell, and a law that imposes the restriction that an agent can only move to a free cell on the grid next to the agent. Reactor effectively applies the influences invoked by the agents. For influences that are related to the application environment, the reactor 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 **Translation** module that converts the high-level influences of agents into

low-level actions in the deployment context.

Communication Service handles the exchange of messages in the agent system. The main functions of the communication service are *mailing* and *message delivering*. Mailing regulates the exchange of messages between agents according to the current state of the application environment and a set of applicable laws. Communication laws (c-laws) impose constraints on the exchange of messages. An example in the Packet-World is a law that drops messages directed to agents outside the communication–range of the sender. In addition, mailing can enforce application–specific regulations on the message stream. For example, mailing can give preferential treatment to high-priority messages. Mailing passes the messages to the translation module that converts the high-level message descriptions (in an agent communication language—ACL) into low-level communication primitives of the deployment context. Translation converts incoming messages from the deployment context into high-level messages descriptions and passes them to message delivering that delivers the messages to the appropriate agents.

Synchronization & Data Processing monitors application-specific parts of the deployment context and keeps the corresponding representation in the state of the application environment up-to-date. An example is the topology of a dynamic network which changes are reflected in a network abstraction maintained in the state of the application environment. The synchronization module typically pre–processes the raw data derived from the deployment context before it passes the data to the state maintenance module, examples are sorting and integration of observed data.

Dynamics maintains activities in the application environment that happen independently of the agents and the deployment context. A typical example is the maintenance of a digital pheromone. In the Packet–World, the environment supports aggregation of digital pheromones (when an agent drops new pheromone the strength of the local pheromone is increased accordingly, reinforcing interesting information) and evaporation (the environment decreases the strength of the pheromones over time, representing truth maintenance).

4.2.2 Model of the Agent

We now direct our attention to the functionalities of agents. The agent model consists of four modules with flows between the modules [206, 228, 227]. Fig. 4.2 shows the model. The modules represent the core functionalities of a situated agent.

The four modules provide the required agent functionalities for the mechanisms of adaptivity we have discussed in chapter 3, including selective perception, action selection with roles and situated commitments, and protocol-based communica-

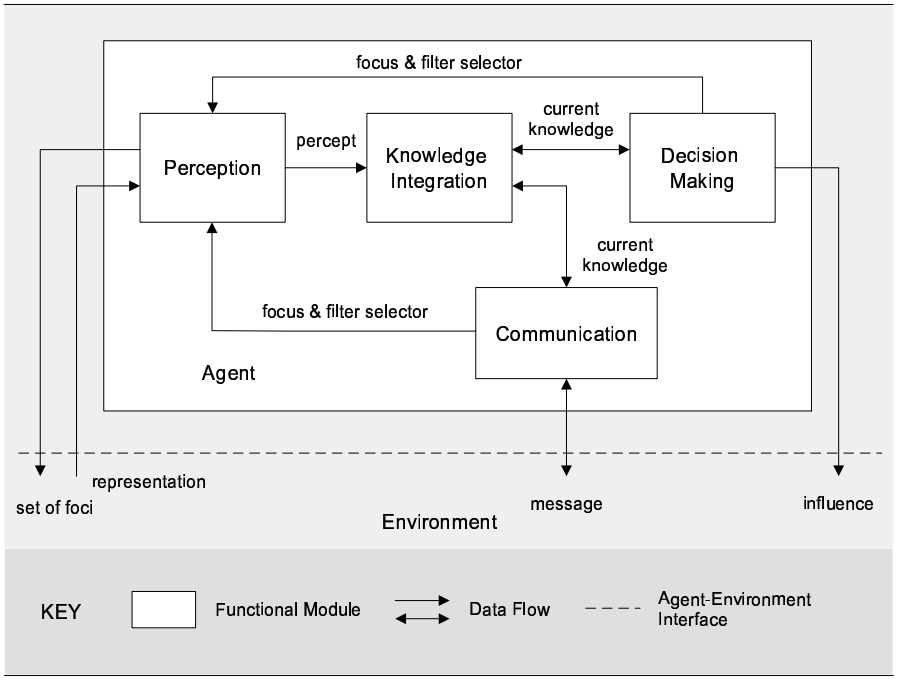


Figure 4.2: Agent model

tion. We explain the four modules in turn.

Knowledge Integration provides the functionality to access and update the agent's current knowledge[1]. The *Perception* module receives perception requests from the *Decision Making* and *Communication* modules to update the agent's knowledge about the environment. Decision making and communication use the agent's current knowledge to make appropriate decisions. Moreover, these modules employ the current knowledge as a means for coordination. For instance, during the progress of a collaboration the communication module modifies the agent's current knowledge implied by the communicative interaction. When a collaboration is established the communication module activates a situated commitment that will affect the agent's action selection in the decision making module. This continues until the situated commitment is deactivated and the collaboration ends. Two cases can be distinguished: the situated commitment can be deactivated due to changes perceived in the environment, or the end of the commitment

[1]We use current knowledge as a synonym for the agent's actual state; see chapter 3, section 3.4.1.

can explicitly be communicated between the collaborating agents. Another example of coordination between decision making and communication is an agent that requests another agent for information that is necessary to complete a task. A simple example is an agent in the Packet-World that has to deliver a packet which destination is outside its perceptual scope. This agent can obtain the location of the destination by requesting it from another agent. The communication module writes the location in the agent's current knowledge which in turn will be used by the decision making module to move the agent efficiently towards the destination.

Perception provides the functionality to an agent for selective perception. Selective perception consists of three basic functions: *sensing*, *interpreting* and *filtering*. Sensing enables the agent to sense the environment with a given set of foci. Foci allow the agent to sense the environment for specific types of information. Examples of foci in the Packet-World are a focus to observe objects (packets, destinations, etc.) and a focus to sense gradient fields. Sensing results in a representation. A representation is a data structure that represents elements in the environment. An example of a representation in the Packet-World is a data structure that describes the sensed objects in an agent's vicinity. Interpreting maps a representation to a percept. The resulting percept describes the sensed environment in the form of knowledge that can be understood by the decision making and communication modules. For example, for a representation that contains a number of packets in a certain area, interpretation may derive a percept with the particular locations of the distinguished packets as well as the location of the group of packets as a cluster. Filtering uses a given set of filters to select only those data items of a percept that match the selection criteria specified by the filters. An example in the Packet-World is a filter that selects the destinations for a particular color of packets. The filtered percept is used by the knowledge integration module to update the agent's current knowledge. 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 provides the functionality to an agent for selecting and invoking influences in the environment. Decision making consists of two basic functions: *influence selection* and *execution*. To select appropriate influences, a situated agent uses a behavior–based action selection mechanism extended with roles and situated commitments. Execution provides the functionality to invoke selected influences in the environment.

Communication provides the functionality to an agent for exchanging messages with other agents according to well-defined communication protocols. Communication consists of three basic functions: *message decoding*, *communicating* and *message encoding*. Message decoding stores incoming messages in a buffer and

decodes the buffered messages one by one. Message decoding extracts the information from a message according to a well defined agent communication language (ACL). A communication language defines the format of the messages, i.e. the subsequent fields the message is composed of. We denote the information extracted from a message as decoded message data. Decoded message data describes the information of a message in a form that can be understood by the agent. The communicating function provides a dual functionality: (1) it interprets decoded message data derived from incoming messages and reacts appropriately; (2) it initiates and continues a conversation when the necessary conditions hold. To perform these tasks, communicating uses a repository of communication protocols, the agent's current knowledge, and an ontology. The ontology defines a shared vocabulary of words that agents use to represent domain concepts and relationships between the concepts. Message encoding enables the agent to encode newly composed message data into messages and passes them to the message delivering system of the environment. To deal with possible delays, this module also provides a buffer.

This concludes the overview of the integrated model for situated multiagent systems. In the next section, we present the reference architecture for situated multiagent systems that maps the functionalities covered by the integrated model onto a system decomposition consisting of software elements and relationships between the elements.

4.3 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 a 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 architec-

ture, 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 module that documents how the module is used in combination with other modules. The interface description distinguishes between provided and required interfaces. A provided interface specifies what functionality the module offers to other modules. A required interface specifies what functionality the 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.

4.3.1 Module Decomposition View Packet 1: Situated Multiagent System

4.3.1.1 Primary Presentation

System	Subsystem
Situated Multiagent System	Agent
	Application Environment

4.3.1.2 Elements and their Properties

A Situated Multiagent System is decomposed in two subsystems: Agent and Application Environment. The subsystems cover the functionality of the modules with the same name in the integrated model for situated multiagent systems, discussed in sections 4.2.1 and 4.2.2.

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.

Application Environment is the medium that 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 is the part of the environment that has to be designed for a concrete multiagent system application. External entities and resources with which the multiagent system interacts are part of the deployment context. The internal structure of the deployment context and its functioning are not considered in the reference architecture.

4.3.1.3 Interface Descriptions

Fig. 4.3 gives an overview of the interfaces of the agent subsystem, the application environment subsystem, and the deployment context.

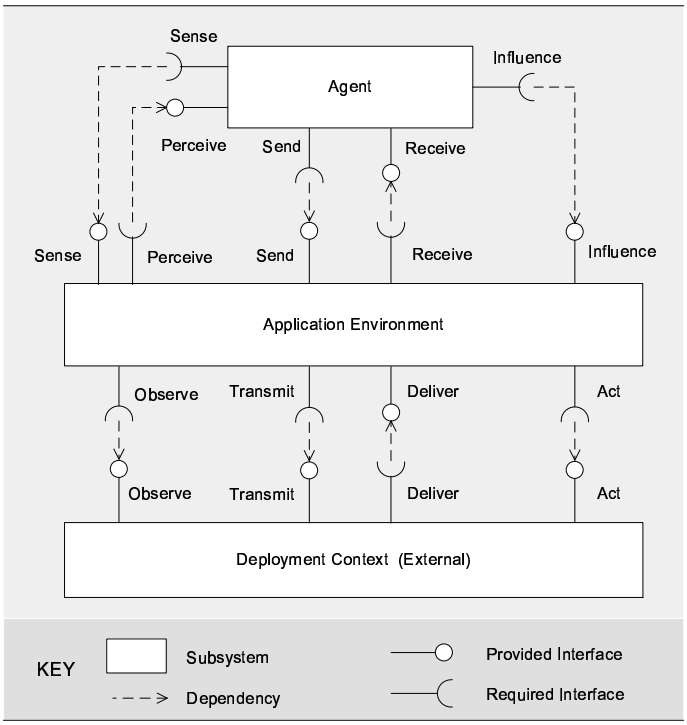


Figure 4.3: Interfaces of agent, application environment, and deployment context

The `Sense` interface enables an agent to sense the environment, `Send` enables an agent to send messages to other agents, and `Influence` enables an agent to invoke influences in the environment. These interfaces are provided by the application environment.

The application environment requires the interface `Perceive` to pass on representations (resulting from sensing the environment) to the agent, and the interface `Receive` to deliver messages. Furthermore, the application environment requires the interface `Observe` from the deployment context to observe particular resources (based on perception requests of agents), `Transmit` to send messages, and `Act` to modify the state of external resources (based on influences invoked by agents).

Finally, the deployment context requires the interface `Deliver` from the application environment to deliver the incoming messages to the agents.

4.3.1.4 Variation Mechanisms

There are three variation mechanisms for this view packet:

M1 *Definition of Agent Types.* A concrete multiagent system application typically consists of agents of different agent types. Agents of the same agent type have the same capabilities and are assigned the same kind of application goals. Each agent type has specific architecture structures. Variations on internal structures for different agent types are discussed in subsequent view packets and views, see sections 4.3.2, 4.4.1, 4.5, and 4.6.

M2 *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 4.4.

M3 *Definition of the Interaction Primitives of the Deployment Context.* To enable the multiagent system 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 to observe resources of the deployment context. An observation primitive indicates which resource is observed and what type of information should be observed.

(2) Action primitives enable to access resources of the deployment context. 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 can be transmitted via the deployment context.

4.3.1.5 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 4.1.2.
- *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 4.1.2.

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.

Distribution. The reference architecture abstracts from the concrete deployment of the multiagent system application, which is highly application dependent. For a distributed application, the deployment context consists of multiple processors deployed on different nodes that are connected through a network. Depending on

the specific application requirements, distribution can take different forms. For some applications, the same instance of the application environment subsystem is deployed on each node. For other applications, specific instances are instantiated on different nodes, e.g., when different types of agents are deployed on different nodes. Some functionalities provided by the application environment may be limited to the local context (e.g., observation of the deployment context may be limited to resources of the local deployment context); other functionalities may be integrated (e.g., neighboring nodes may share state). Integration of functionality among nodes typically requires additional support. Such support may be provided (partially or complete) by appropriate middleware. Examples are support for message transfer in a distributed setting (e.g. [38]), support for a distributed pheromone infrastructure (e.g. [52]), and support for mobility (e.g. [116]). If the required support is not available, it has to be developed and integrated with the rest of the application logic.

Distribution is an important concern in many application domains. Introducing distribution in the reference architecture would result in a more specific architecture that is tailored toward such applications. The proposed reference architecture, however, abstracts from distribution. As such, the reference architecture is also valuable for non-distributed applications, such as simulations. In chapter 5, we discuss an automated transportation system and show how distribution can be integrated with the functionality provided by the reference architecture when designing a concrete software architecture for a distributed application.

Other Concerns. 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 [79]. 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 [79] 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. [24, 191, 64].

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.

4.3.2 Module Decomposition View Packet 2: Agent

4.3.2.1 Primary Presentation

Subsystem	Module
Agent	Perception
	Decision Making
	Communication

4.3.2.2 Elements of the View

The Agent subsystem is decomposed in three modules: Perception, Decision Making and Communication. The modules cover the functionality of the modules with the same name in the integrated model for situated agent discussed in section 4.2.2.

Perception is responsible for collecting runtime information from the environment (application environment and deployment context). The perception module supports selective perception [224, 228]. 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.

Decision Making is responsible for action selection. The action model of the reference architecture is based on the influence–reaction model introduced in [76]. 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 [206].

To enable situated agents to set up collaborations, behavior-based action selection mechanisms are extended with the notions of role and situated commitment [225, 226, 185, 184, 227]. A role represents a coherent part of an agent's functionality in the context of an organization. A situated commitment is an en-

gagement 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 [227]. 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[2] 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 4.3.1.

4.3.2.3 Interface Descriptions

The interface descriptions specify how the modules of an agent are used with one another, see Fig. 4.4. The interfacing with the data repositories is discussed in section 4.4.1.

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

[2] A performative is a word that incites an addressee to perform a particular action [25]. Examples of performatives are *request*, *propose*, and *accept*.

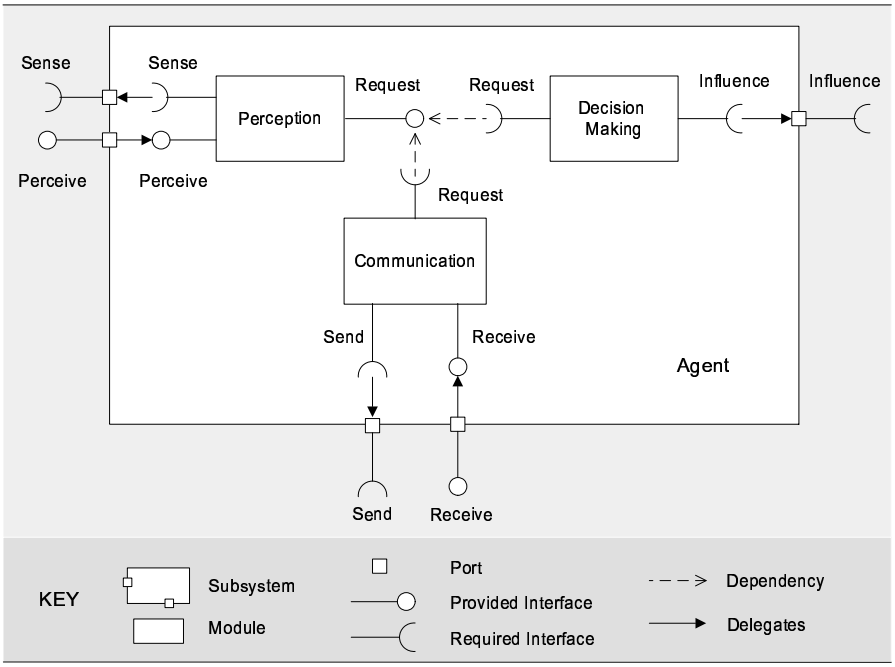


Figure 4.4: Interfaces of the agent modules

environment selectively.

The provided interfaces of agent, `Perceive` and `Receive`, delegate for processing to the provided `Perceive` interface of the perception module and the provided `Receive` interface of the communication module respectively. 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. 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.

4.3.2.4 Variation Mechanisms

This view packet provides the following variation mechanisms:

M4 *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.

M5 *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.

M6 *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 M2 in section 4.3.1.4.

M7 *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.

M8 *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.

M9 *Definition of Roles and Situated Commitments.* 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. Further details of the specification of roles and situated commitments are discussed in subsequent view packets.

M10 *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 M2 in section 4.3.1.4.

The definition of protocols is discussed in the collaborating component view, see section 4.5.

4.3.2.5 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. However, the two modules do not act independently of one another, on the contrary. Decision making and communication coordinate to complete the agent's tasks more efficiently. 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, as listed above (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.

4.3.3 Module Decomposition View Packet 3: Application Environment

4.3.3.1 Primary Presentation

Subsystem	Module
Application Environment	Representation Generator
	Observation & Data Processing
	Interaction
	Communication Service
	Translation
	Synchronization & Data Processing
	Dynamics

4.3.3.2 Elements and their Properties

The Application Environment subsystem is decomposed in seven modules. The modules cover the functionality of the modules with the same name in the integrated model for application environment, discussed in section 4.2.1. 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. 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 collected from resources in the deployment context is returned to the requester, i.e. the representation generator. Rather than delivering raw data retrieved from the resources in the deployment context, the observation & data processing module can provide additional functions to preprocess data, examples are sorting and integration of sensor 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. Agents influences are subject to action laws that represent domain specific constraints on agents' influences. 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 translation module that converts the influences invoked by the agents into low-level action primitives in the deployment context.

The **Communication Service** is responsible for collecting messages; it provides the necessary infrastructure to buffer messages, and to deliver messages to the appropriate agents. The communication service 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. To actually transmit the messages, the communication service makes use of a (distributed) message transfer system provided by the deployment context. The communication service uses the translation module to convert the high-level message descriptions into low-level communication primitives of the deployment context.

Translation bridges the gap between influence and message descriptions used by agents and the corresponding action and communication primitives of the deployment context. Influences and messages used by agents are typically described at a higher-level of abstraction. For example, a FIPA ACL message [77] 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.

Translation provides a dual functionality: (1) it translates influences into low-level action primitives with the deployment context; and (2) it translates ACL messages into low-level formatted messages (that can be transmitted via the deployment context) and vice versa.

Synchronization & Data Processing monitors domain-specific parts of the deployment context and keeps the corresponding representation in the state of the application environment up-to-date. 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.

Dynamics is responsible for maintaining processes in the application environment that happen independent of agents and the deployment context. The dynamics module directly accesses the state of the application environment and maintains this state according to its application specific definition (see the shared data view in section 4.4.2).

4.3.3.3 Interface Descriptions

The interface descriptions specify how the modules of the application environment are used with one another, see Fig. 4.5. The interfacing with data repositories of the application environment is discussed in section 4.4.2.

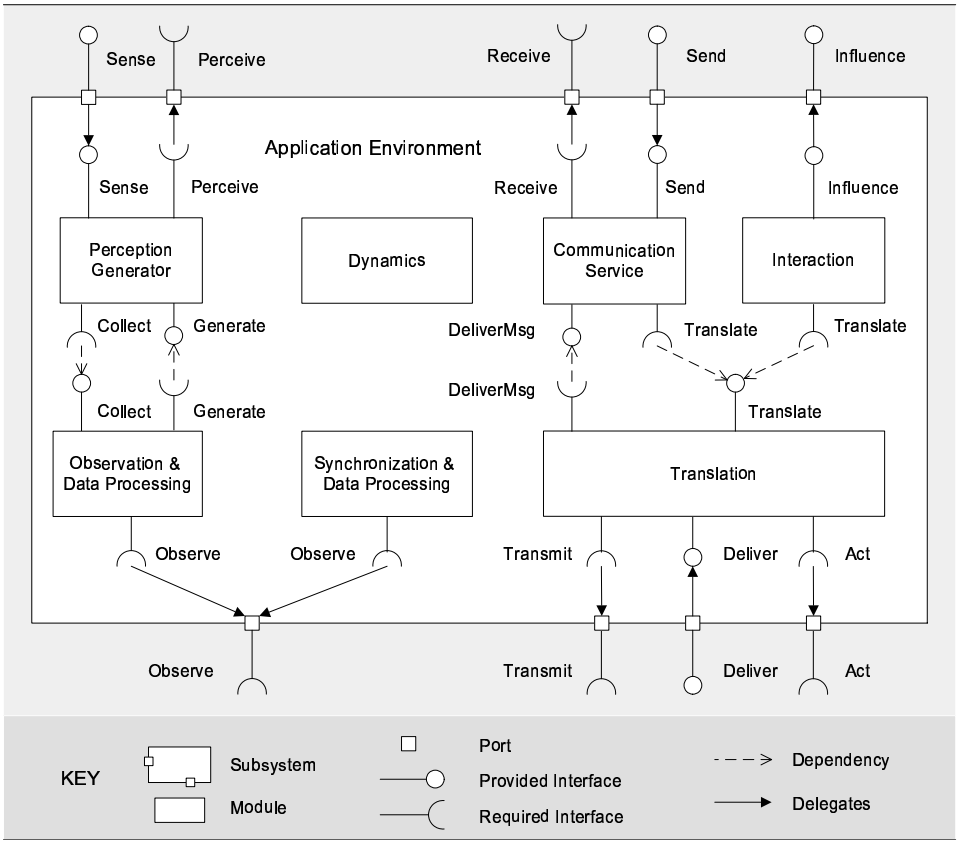


Figure 4.5: Interfaces of the application environment modules

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 `Collect` depends on the `Collect` 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 data that results from the observation of resources in the deployment context is processed by the observation & data processing module. The `Generate` interface of observation & data processing uses the perception generator's provided interface `Generate` to generate a representation for the requesting agent based on the processed data. The `Perceive` interface of the perception generator delegates the delivering of the perception result (i.e. a representation of the sensed elements) to the `Perceive` interface of the application environment that makes the representation available to the agent.

The only interface required by the synchronization & data processing module for its functioning (i.e. synchronize the state of the application environment with particular resources in the deployment context) is `Observe`. The processing of this interface is delegated to the `Observe` interface of 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 `Send` interface of the communication service. To convert messages into a low-level format for transmission via the deployment context, the communication service's required interface `Translate` depends on the interface `Translate` provided by the translation module. The `Transmit` interface of the translation module 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 translation module. Translation converts the messages into an appropriate format for agents and uses the `DeliverMsg` interface of the communication service to deliver the messages. The `Receive` interface of the communication service 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 `Translate` depends on the interface `Translate` provided by the translation module. This latter interface provides the functionality to convert influences into low-level action primitives of the deployment context[3]. The `Act` interface of the translation module delegates the actions to external resources to the `Act` interface of the application environment that invokes the actions in the deployment context.

Notice that the dynamics module does not provide or require any function-

[3]Actually, the application environment employs an internal representation of influences, we explain the details in the collaborating components view in section 4.5.2.

ality of the other modules of the application environment. The interfacing of dynamics with the state repository of the application environment is discussed in section 4.4.2.

4.3.3.4 Variation Mechanisms

This view packet provides the following variation mechanisms:

M11 *Omission of Observation, Synchronization, and Translation.* For applications that do not interact with external resources, the observation, synchronization, and translation modules can be omitted. For such applications, the environment is entirely virtual and as such only consists of the application environment.

M12 *Omission of Communication Service.* For agent systems in which agents do not communicate via message exchange, the communication module can be omitted, see also variation mechanism M4 in section 4.3.2.

M13 *Omission of Dynamics.* For multiagent system applications with an application environment that does not have to maintain dynamics independent of agents, the dynamics module can be omitted.

M14 *Definition of Observations.* Observations enable the multiagent system to collect data from resources in 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 collaborating component view, see section 4.5.

4.3.3.5 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. 4.6.

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. Besides influences invoked by

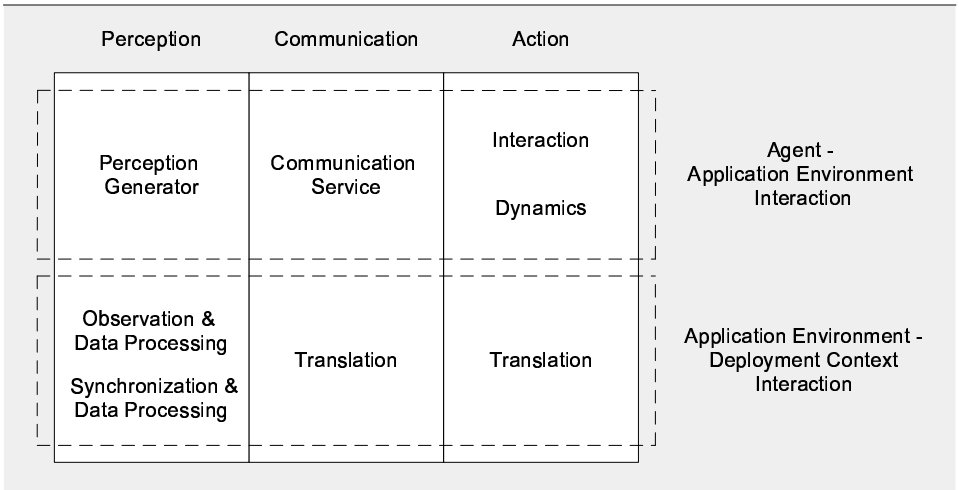


Figure 4.6: Decomposition application environment

agents, we also consider activities that happen independent of agents and that modify the state of the application environment as part of the action column.

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 service, and interaction and dynamics. 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, and translation. 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.

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 (the translation module spans two cells, proving the functionality for the translation of influences and messages.). 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.

4.4 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 [60]. 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).

4.4.1 C & C Shared Data View Packet 1: Agent

4.4.1.1 Primary Presentation

The primary presentation is shown in Fig. 4.7.

4.4.1.2 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 4.3.2. The data accessors share the Current Knowledge repository. This repository provides the functionality of the Knowledge Integration module of the agent model, discussed in section 4.2.2.

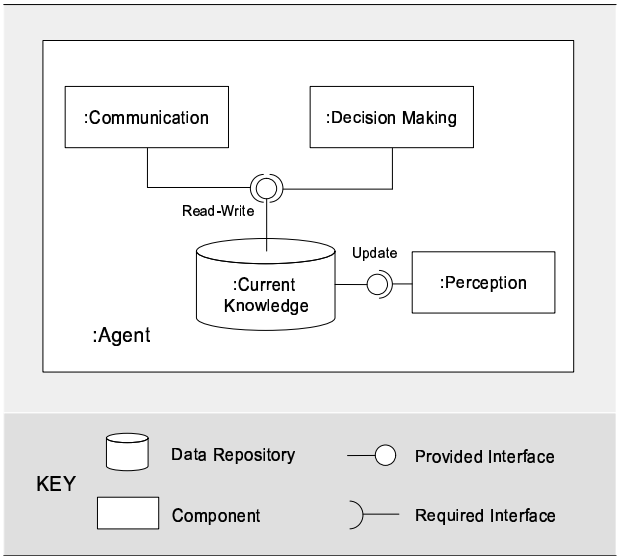


Figure 4.7: Shared data view of an agent

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 4.3.2.

4.4.1.3 Interface Descriptions

Fig. 4.7 shows the interconnections between the current knowledge repository and the internal components of the agent. These interconnections are called assembly connectors [13]. 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.

4.4.1.4 Variation Mechanisms

This view packet provides one variation mechanism:

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 M2 in section 4.3.1.4. 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.

4.4.1.5 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 4.3.2.5).

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.

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

4.4.2.1 Primary Presentation

The primary presentation is depicted in Fig. 4.8.

4.4.2.2 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 4.3.3. The State repository provides the functionality of the State Maintenance module of the environment model, discussed in section 4.2.1. The Laws repository encapsulates the various laws (perception laws, communication laws, and action laws) of the environment model.

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

To perform their functionalities, interaction, dynamics, synchronization & data processing, and observation & data processing can read and write state of the application environment. The representation generator, the communication service, and the translation components 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.

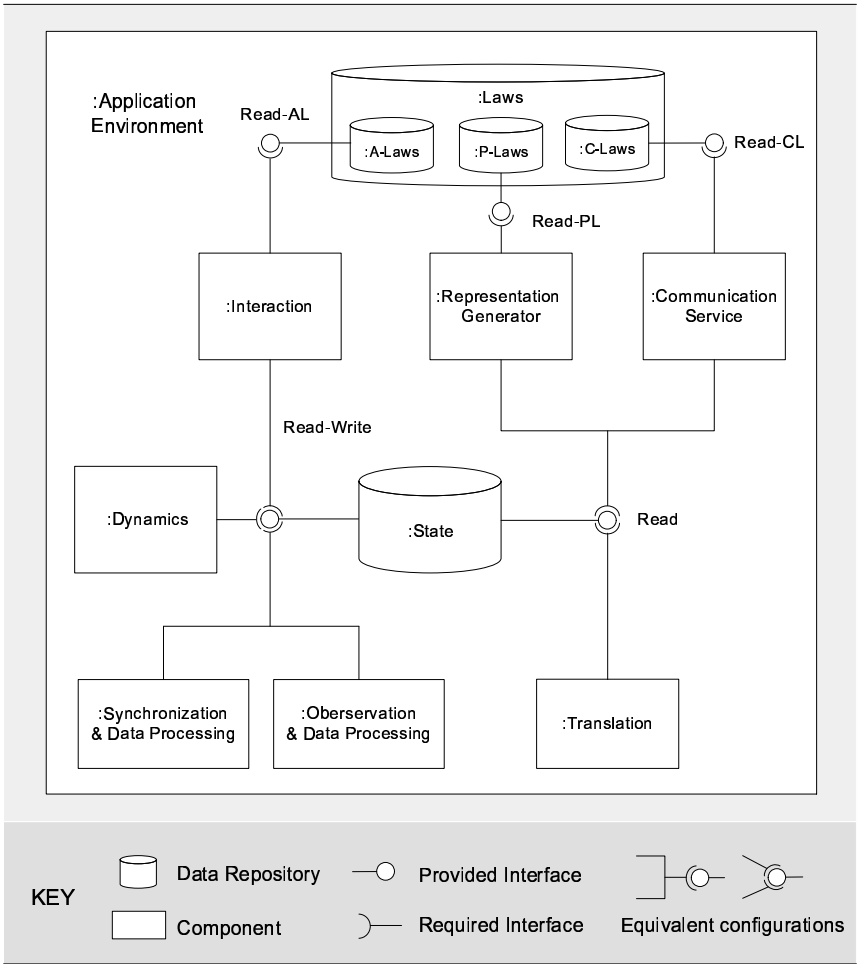


Figure 4.8: Shared data view of the application environment

4.4.2.3 Interface Descriptions

Fig. 4.8 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.

4.4.2.4 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 M2 in section 4.3.1.4. 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.

The variation mechanisms for the various laws of the application environment is discussed in the collaboration components view, see section 4.5.

4.4.2.5 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.

4.5 Component & Connector Collaborating Components View

The collaborating components view shows the multiagent system as a set of interacting runtime components that use a set of shared data repositories to realize the required system functionalities. We have introduced the collaborating components view to explain how collaborating components realize various functionalities in the multiagent system. The elements of the collaborating components view are:

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

The collaborating components view is an excellent vehicle to learn the runtime behavior of a situated multiagent system. The view shows the data flows between runtime components and the interaction with data stores, and it specifies the functionalities of the various components in terms of incoming and outgoing data flows. Each view packet in this view zooms in on one coherent part of system functionality. A view packet shows how a number of components collaborate to realize that particular functionality in the system.

The reference architecture provides three view packets of the collaborating components view. We start with the view packet that describes the collaborating components of perception, i.e. the perception component and the representation generator. Next, we discuss the view packet that describes the collaborating components of interaction, i.e. decision making and interaction. Finally, we discuss the view packet of communication that describes the collaboration between the communication component and communication service.

4.5.1 C&C Collaborating Components View Packet 1: Perception and Representation Generator

4.5.1.1 Primary Presentation

The primary presentation is shown in Fig. 4.9.

4.5.1.2 Elements and their Properties

This view packet shows how a number of collaborating components realize the functionality for perception. The elements in this view packet are the components of perception and representation generator and local repositories. To explain the

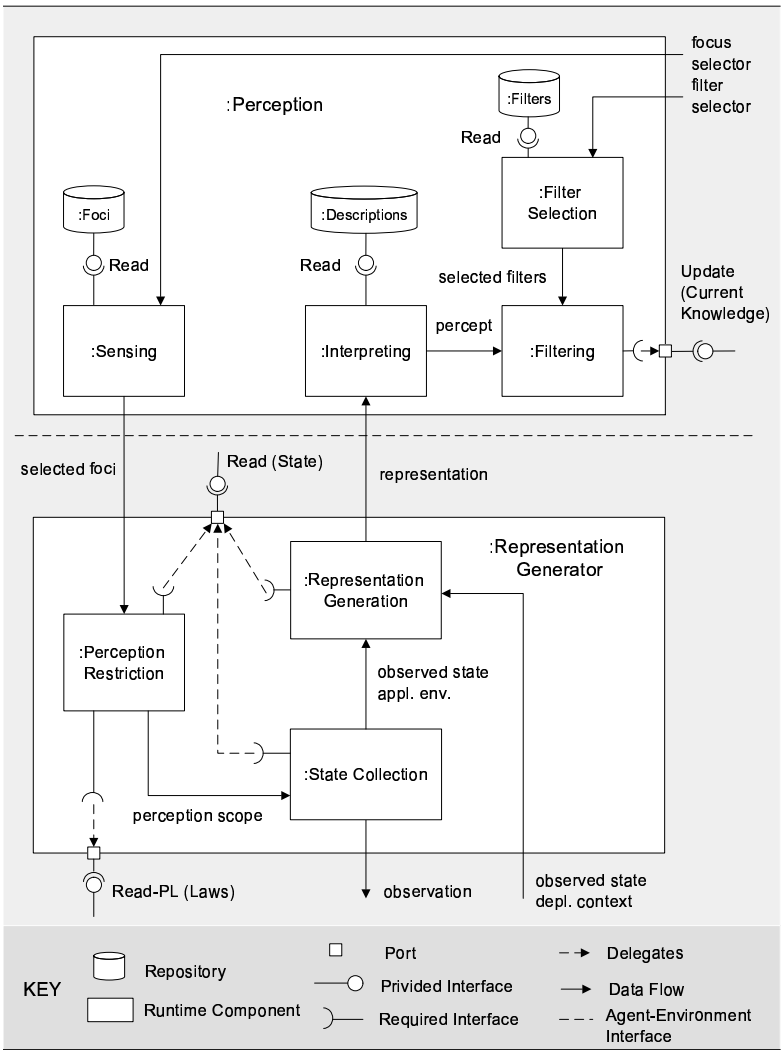


Figure 4.9: Collaborating components of perception and representation generator

collaboration between the various components, we follow the logical thread of successive activities that take place from the moment an agent takes the initiative to sense the environment until the percept is available to update the agent's current knowledge.

`Sensing` takes a focus selector and the set of foci of the agent, and selects the corresponding foci to produce a perception request. `PerceptionRestriction`

takes the set of foci of a perception request, the current state of the application environment, and generates according to the set of perception laws, a perception scope. A perception scope delineates the scope of perception for the perception request according to the constraints imposed by the perception laws. For example, for `request(i,sense-objects())` which is a perception request of an agent in the Packet–World with identity `i` that senses the visible objects in its vicinity, a possible perception scope is `visibleObjects(agent(i),position(3,6),5)`. In this particular example, the number `5` refers to the default sensing range for visible perception of five cells.

`StateCollection` collects the observable state for the perception scope, given the current state of the environment. In particular, state collection selects the subset of state elements of the application environment for the given perception scope (observed state appl. env.), and produces—if applicable—an observation to collect data from the deployment context within the scope of perception. `RepresentationGeneration` takes the observed state of the application environment together with the state observed from the deployment context (observed state depl. context) and produces a representation.

`Interpreting` uses the agent's set of descriptions to interpret the given representation. Interpreting results in a percept for the agent. `FilterSelection` selects a subset of filters from the agent's set of filters according to the given filter selector. `Filtering` filters the percept of the agent according to the set of selected filters. Finally, the filtered percept is used to update the agent's current knowledge.

4.5.1.3 Variation Mechanisms

This view packet provides two variation mechanisms:

CC1 *Definition of Descriptions.* Descriptions enable an agent to interpret representations derived from the observation of the environment. A description can be defined as a template that specifies a particular pattern of a representation. Interpreting a representation than comes down to searching for matches between the description template and the examined representation. Each match yields data of a percept that is used to update the agent's state, possibly after some preceding filtering.

CC2 *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 network can be defined as a function of the actual traffic on the network.

4.5.1.4 Design Rationale

The integrated set of components of perception and representation generator provide the functionality for selective perception in the situated multiagent system. The overall functionality results from the collaboration of the various components. In this collaboration, each component provides a clear-cut functionality, while the coupling between the component is kept low. Concepts such as foci, descriptions, filters, and laws are first-class in the reference architecture. This helps to improve modifiability an reusability.

Selective perception allows an agent to adapt its perception according to its current tasks. The reference architecture supports adaptation of perception laws according the changing circumstances in the environment. Both these properties contribute to the flexibility of the system.

4.5.2 C&C Collaborating Components View Packet 2: Decision Making and Interaction

4.5.2.1 Primary Presentation

The primary presentation is shown in Fig. 4.10.

4.5.2.2 Elements and their Properties

This view packet shows the collaborating components that realize the functionality for action. The elements in this view packet are the components of decision making and interaction and two local repositories. For the discussion of the collaborating components, we follow the successive activities that take place from the moment an agent initiates decision making until the effects of the selected influence are completed.

`D-KnowledgeUpdate` enables the agent to update its current knowledge of the environment. D-KnowledgeUpdate takes the agent current knowledge and the set of *current stimuli* and produces a focus and filter selector. We distinguish between two types of stimuli: external and internal stimuli. An external stimulus refers to a factor in the environment that drives the agent's decision making (an example is a gradient field that guides an agent towards a particular location). An internal stimulus is an internal factor that affects an agent's decision making (an example is the agent's available energy to act in the environment). The focus and filter selector is passed to the perception module that senses the environment to produce a new percept and update the agent's knowledge.

`ActionSelection` takes the current stimuli and the knowledge of the agent to select an *operator* and update the set of current stimuli. An operator is an internal representation used by the agent to represent the selected action. Decision making converts the operator into an influence that is invoked in the environment, see the discussion of *Execution* below.

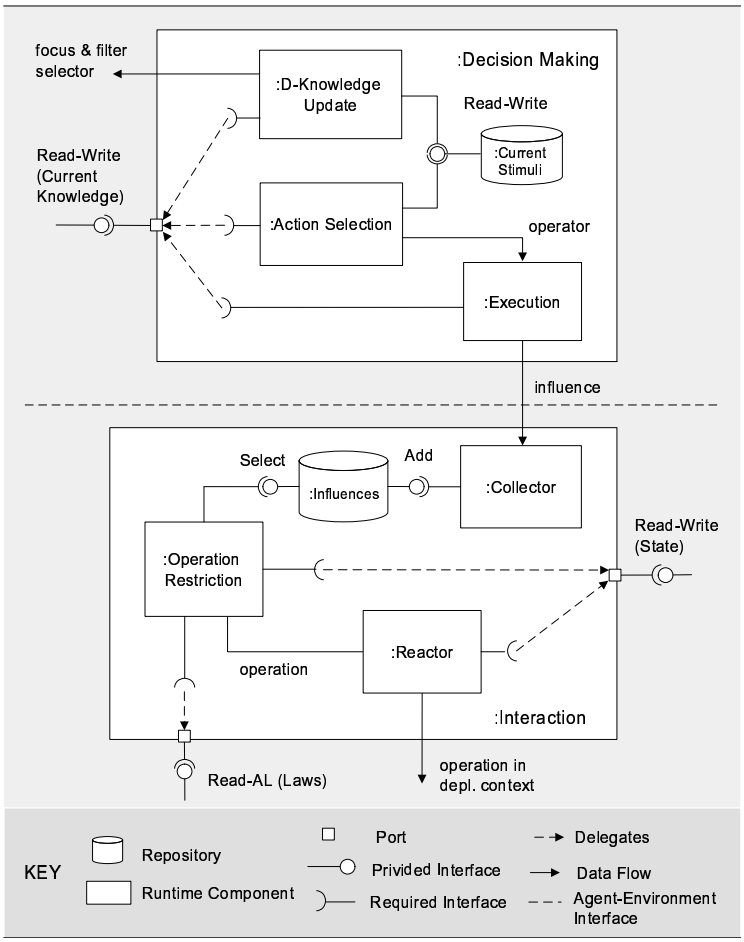


Figure 4.10: Collaborating components of decision making and interaction

To select suitable operators, the action selection component encapsulates a behavior-based action selection mechanism. In general, a behavior–based action selection mechanism consists of a set of behavior modules. Each behavior module is a relatively simple computation module that tightly couples sensing to action. An arbitration or mediation scheme controls which behavior-producing module has control and selects the next action of the agent. To enable situated agents to set up collaborations, behavior-based action selection mechanisms are extended with the notions of role and situated commitment. Behavior modules that represent a coherent part of an agent's functionality in the context of an organization are denoted as a role. A role is defined as a 5-tuple:

$$\Re : \langle name, stimuli, operators, select \rangle$$

Roles have a well-known *name* that is shared among agents in the system. The function *select* maps the set of *stimuli* to a set of $\langle operator, pref\text{–}factor \rangle$ tuples, one for each operator of the set of operators that can be selected by the role. A *pref–factor* determines the relative preference for selecting the accompanying operator. The $\langle operator, pref\text{–}factor \rangle$ tuples are used by the arbitration schema to determine which role has control and which operator is selected for execution.

A situated commitment is defined as a 7-tuple:

$$C : \langle name, rel\text{–}set, context, act\text{–}con, deact\text{–}con, status, rolemap \rangle$$

As for roles, situated commitments have a well-known *name*. Explicitly naming roles and commitments enables agents to set up collaborations, reflected in mutual commitments. The relations set *rel–set* contains the identity of the related agent(s) in the situated commitment. The *context* describes contextual properties of the situated commitment such as descriptions of objects in the local environment. *act–con* and *deact–con* are the activation and deactivation conditions that determine the *status* of the situated commitment. When the activation condition becomes true, the situated commitment is activated. The behavior of the agent will than be biased according to the specification of the *rolemap*. The *rolemap* specifies the relative weight of the preference factors of the operators of different roles. In its simplest form, the *rolemap* narrows the agent's action selection to operators in one particular role. An advanced example is a *rolemap* that biases the operator selection towards the operators of one role relative to the preference factors of operators of a number of other roles of the agent. As soon as the deactivation condition becomes true, the situated commitment is deactivated and will no longer affect the behavior of the agent.

`Execution` takes the selected operator and the current knowledge of the agent and produces an influence that is invoked in the environment. Execution decouples the agent's internal representation of activity in the environment from the influences that are available to the agent to access and modify the state of the environment.

The `Collector` collects influences invoked by the agents, and adds the influences to the set of pending influences in the agent system. `OperationRestriction` selects an influence from the set of pending influences and converts the influence into an *operation*. An operation is a representation of the selected influence internal to the application environment. For example, for `influence(i, step(North))` which is an influence produced by an agent in the Packet–World with identity `i` that attempts to make a `step` on the grid in the direction `North`, a possible operation is `move(agent(i),position(3,6),destination(2,6))`. The selection of influences is based on an influence selection policy that specifies the ordering of influences, taking into account the current state of the environment. OperationRestriction applies the set of action laws to the selected operation, given the current

state of the environment. The action laws impose restrictions on the kind of manipulations agents can perform in the environment.

The `Reactor` applies the operation to the current state of the application environment and produces an operation to act in the deployment context if applicable. The execution of the operation modifies the state of the application environment and produces an operation in the deployment context that is passed to the translation module.

4.5.2.3 Variation Mechanisms

There are three variation mechanisms for this view packet:

CC3 *Specification of the Action Selection Mechanism.* Situated agents use a behavior-based action selection mechanism. For the chosen action selection mechanism the operators and the various stimuli have to be specified. Furthermore, the definition of the agent's roles and situated commitments have to be completed, see also variation mechanisms M9 in section 4.3.2.4. For each role, the *select* function has to be defined (see the definition of a role above). For each situated commitment, the *relation set*, the *activation* and *deactivation conditions*, and the *rolemap* have to be defined (see the definition of situated commitment above). Finally, the roles and situated commitments have to be integrated with the arbitration mechanism of the selected behavior-based action selection mechanism. An example specification of a behavior-based action selection mechanism that integrates roles and situated commitments in a free-flow tree is discussed in chapter 3, see section 3.4.3.2.

CC4 *Definition of the Influence Selection Policy.* An influence selection policy imposes an application specific ordering on the execution of pending influences. Simple policies are first-in-first-out and random selection. Advanced forms of influence ordering can be defined based on a combination of criteria, such as the types of influences, the types of agents that have invoked the influences, and even the identities/priorities of the invoking agents.

CC5 *Definition of Operations.* The application environment uses operations as internal representations for the influences invoked by the agents. Operations can represent influences that target the manipulation of the state of the application environment, or influences that target the manipulation of elements in the deployment context. The concrete definition of operations is closely related to the definition of the state of the application environment, see variation mechanism SD2 in section 4.4.2.4.

CC6 *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.

4.5.2.4 Design Rationale

The collaborating components of decision making and interaction provide the functionality for action execution in the agent system. In this collaboration, each component provides a clear-cut functionality, while the coupling between the components is kept low. This helps to improve modifiability and reusability.

Behavior–based action selection enables agents to behave according to the situation in the environment, and change their behavior with changing circumstances. The notions of a role and situated commitment enable agents to set up collaborations. The duration of a situated commitment can be regulated based on conditions in the local context in which the collaborating agents are placed. This approach fits the general principle of situatedness and improves flexibility and openness. An agent adapts its behavior when the conditions in the environment change or when agents enter or leave its scope of interaction.

Action laws and the influence selection policy provide a means to regulate the actions of the agents in the system. Both action laws and the influence selection policy can be defined in terms of the actual conditions in the environment, contributing to the flexibility of the system.

4.5.3 C&C Collaborating Components View Packet 3: Communication and Communication Service

4.5.3.1 Primary Presentation

The primary presentation is shown in Fig. 4.11.

4.5.3.2 Elements and their Properties

This view packet shows how a number of collaborating components realize the functionality for communication. The elements in this view packet are the components of communication and communication service, and local repositories. Similarly as in the previous view packets of the collaboration components view, we discuss the collaboration between the components by following the successive activities that take place from the moment an agent takes the initiative to send a message until the addressees have received the message and can react to it.

We start with `Communicating` that handles the agent's communicative interactions. The communicating component processes incoming messages, and produces

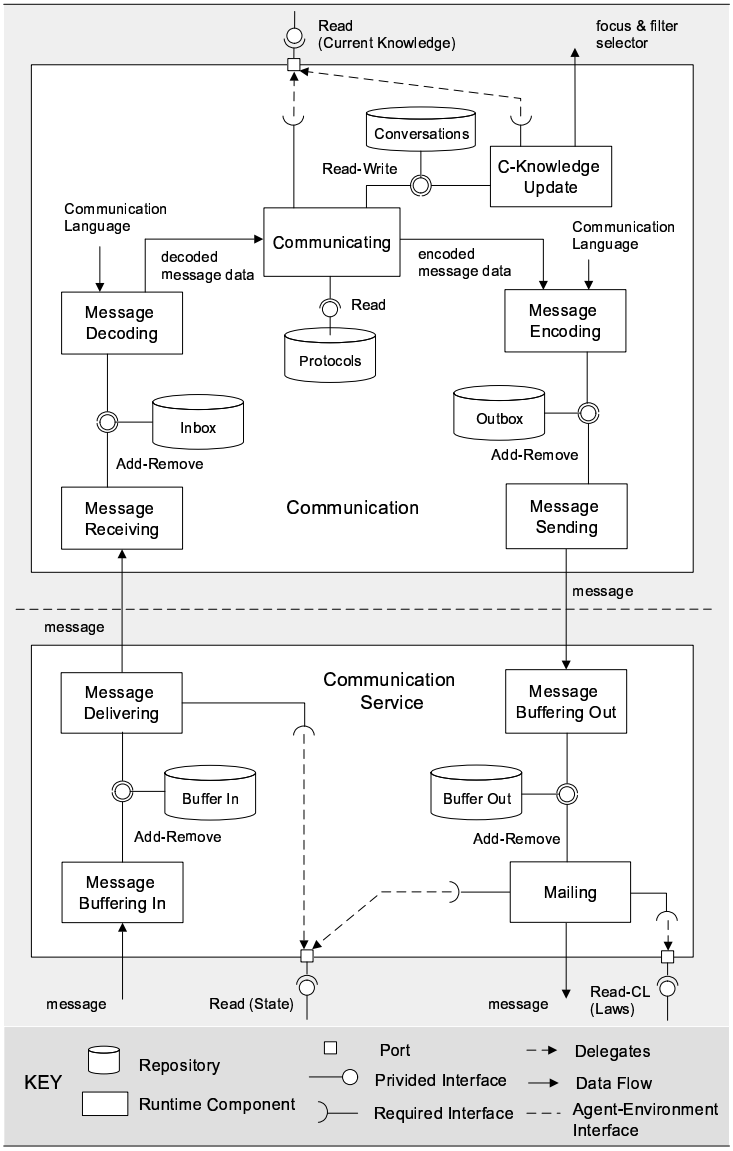


Figure 4.11: Collaborating components of communication and communication service

outgoing messages according to well-defined communication protocols. Communicative interactions can modify the agent's state possibly affecting the agent's

selection of influences; a typical example is the activation and deactivation of situated commitments. A communication protocol consists of a series of protocol steps. A protocol step is a tuple $\langle condition, effect \rangle$. Condition is a boolean expression that determines whether the step is applicable. Expressions can take into account the agent's current knowledge, the actual set of conversations that contains the agent's history of ongoing interactions, the set of available protocols, and possibly the decoded message data of a received message. We distinguish between three types of protocol steps: conversation initiation, conversation continuation, and conversation termination. A conversation initiation step starts a new conversation according to a particular protocol. An agent initiates a conversation based on its current knowledge, possibly taking into account the decoded message data of a message received from another agent that started the interaction. A conversation continuation performs a step in an ongoing conversation. A conversation continuation may deal with a received message without directly responding to it, it may immediately react with a reply message, or it may pick up a conversation after a break. Finally, a conversation termination concludes an ongoing conversation. The termination of a conversation can be induced by changing circumstances in the environment or it can directly result from a preceding step of the conversation.

`C-KnowledgeUpdate` enables the agent to update its current knowledge according to its ongoing conversations. The knowledge update component takes the agent current knowledge and the set of conversations and produces a focus and filter selector that is passed to the perception module that senses the environment to produce a new percept and update the agent's knowledge.

`MessageEncoding` encodes newly composed message data into messages and puts the messages in the outbox buffer of the agent. Message encoding is based on the communication language that is shared among the agents in the system. `MessageSending` selects a message from the set of pending messages in the outbox buffer and passes it to the communication service.

`MessageBufferingOut` collects messages sent by agents and puts them in the output buffer of the communication service. `Mailing` selects a message from the output buffer given a message selection policy, and applies the set of communication laws to the selected message taking into account the current state of the environment. To determine the order in which messages are sent, the selection policy takes into account the current state of the environment. Mailing passes the messages to the translation component that handles the conversion of the messages for transmission.

`MessageBufferingIn` collects incoming messages and puts them in the input buffer of the environment. `MessageDelivering` delivers the messages of the input buffer to the appropriate agents.

`MessageReceiving` accepts messages and puts them in the agent's inbox. Finally, `MessageDecoding` selects a message from the agent's inbox and decodes the message according to the given communication language. The decoded message

data of the selected message is passed to the communicating component that will process it.

4.5.3.3 Variation Mechanisms

There are three variation mechanisms in this view packet.

CC7 *Definition of Communication Protocols.* The definition of a concrete communication protocol includes the specification of (1) the conditions for the agent to initiate the protocol, (2) the conditions to continue the interaction in the various stages of the protocol, (3) the conditions to terminate the interaction. For each situation, the protocol has to define the concrete actions that have to be executed. Actions include the processing of received messages, the composition of new messages, and the update of the current knowledge. An important aspect of this latter is the activation/deactivation of situated commitments. State charts [87, 13] are one possible approach to specify a communication protocol; an interesting example is discussed in [67].

CC8 *Definition of the Message Selection Policy.* A message selection policy imposes an application specific ordering on the transmission of pending messages from the output buffer of the application environment. A simple policy is first-in-first-out. Advanced forms of ordering can be defined based on a combination of criteria, such as ordering based on the type of messages and the type of agents that have sent the messages. In addition, the policy can take into account runtime information such as the current load of the network to select messages.

In principle, a selection policy has to be defined for the input buffer of the communication service as well. However, since the message stream between agents is already regulated by the message selection policy of the output buffer, a simple static selection policy such as first-in-first-out can be used for selecting messages from the input buffer. Usually, a simple policy is used for the selection of messages from the agent's inbox and outbox buffers as well. Although this might cause suboptimal behavior in particular situations, in general it avoids complex and time-consuming analysis of the collections of buffered messages.

CC9 *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.

4.5.3.4 Design Rationale

The collaborating components of communication and communication service provide the functionality for message exchange in the agent system. Direct communication allows situated agents to exchange information and set up collaborations. Coordination through message exchange is complementary to mediated coordination via marks in the environment (e.g. pheromone-based coordination). The various components in the collaboration are assigned clear-cut responsibilities and coupling amongst components is kept low.

Communication defined in terms of protocols puts the focus of communication on the relationship between messages. In each step of a communicative interaction, conditions determine the agent's behavior in the conversation. Conditions not only depend on the status of the ongoing conversations and the content of received messages, but also on the actual conditions in the environment reflected in the agent's current knowledge and in particular on the status of the agent's commitments. This contributes to the flexibility of the agent's behavior.

The communication service only provides basic functionality for message exchange. Direct communication in a situated multiagent system should comply to the principle of locality, i.e. communication should serve as a means for agents to exchange information and set up collaborations with agents in their (logical) neighborhood. If necessary, the communication service can be provided with additional services, such as a yellow–page service to enable agents to find particular service providers. But, since such services are uncommon for situated agent systems, they are not included in the reference architecture.

The message selection policy and communication laws provide mechanisms to regulate the message stream in the agent system. Both policies and laws are imposed according the changing conditions in the environment, contributing to the flexibility of the system.

4.6 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 [60].

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.

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

4.6.1.1 Primary Presentation

The primary presentation is shown in Fig. 4.12.

4.6.1.2 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.

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 independent 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. State collection is subject to the perception laws. The observation process returns the observed data to the representation generator process as soon as it becomes available. 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

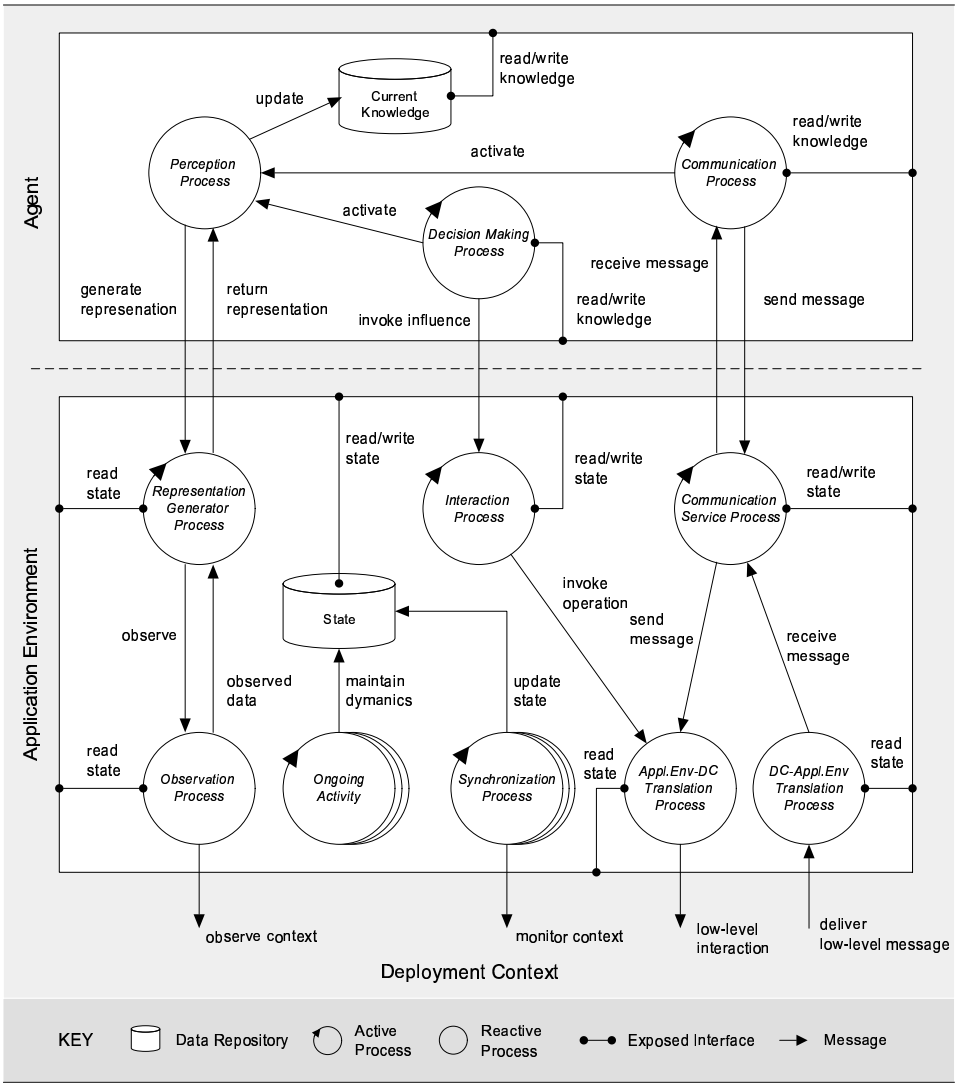


Figure 4.12: Communicating processes view for perception, interaction, and communication

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 `Appl.Env-DC Translation Process`. This translation 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 Service Process` that applies the communication laws and subsequently passes the messages to the `Appl.Env-DC Translation Process`. This latter process converts the messages into low–level interactions with the deployment context. Furthermore, the `DC-Appl.Env Translation 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 service process. The communication process delivers the messages to the communication process of the appropriate agent.

Independent Processes in the Application Environment. The `Synchronization Processes` are active processes that monitor application specific parts of the deployment context and keep the corresponding state of the application environment up-to-date. `Ongoing Activities` are active processes that maintain application specific dynamics in the application environment.

4.6.1.3 Variation Mechanisms

There are two variation mechanisms in this view packet.

CP1 *Definition of State Synchronization with the Deployment Context.* 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.

CP2 *Definition of Ongoing Activities.* For each ongoing activity in the application environment an active process has to be defined. Processes of ongoing activities may run independently, or they may monitor and react to particular events in the application environment. Examples of the former are a clock, and the evaporation of digital pheromones. Examples of the latter are a timer that is triggered by a particular event, and the diffusion of new pheromone that is dropped by an agent.

4.6.1.4 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 translation processes are also reactive, they provide their services on command of other processes. Finally, synchronization processes and ongoing activities 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. Ongoing activities represent dynamics in the application environment that happen independent of agents and the deployment context. These processes are responsible to maintain the state the application environment according to the ongoing dynamics.

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.

4.7 A Framework that Implements the Reference Architecture

To demonstrate the feasibility of the reference architecture, we have developed an object-oriented framework that implements the main modules of the reference architecture for situated multiagent systems. Appendix B gives an overview of the framework and illustrates how the framework is specialized for the Packet-World and an experimental robot application.

The framework shows a concrete design of the reference architecture. It supports the development of situated agent systems with a software environment as well as systems with a physical environment. Support is provided for selective perception, protocol-based communication, behavior-based decision making, and dynamics that happen independent of agents' activities. Interaction in the environment is extended with support for simultaneous actions. Simultaneous actions are actions that happen together and that can have a combined effect in the application environment [203, 207, 205]. An example of simultaneous actions in the Packet–World are two agents that push the same packet in different directions. As a result, the packet moves according to the resultant of the two actions. Similar to the reference architecture, the framework offers support for laws that enable the designer to put constraints on the various kinds of activities of agents in the

environment. The framework provides no support for distribution of a software environment and interaction with the deployment context.

Developing the framework was a valuable experience. It has improved our general understanding of important aspects of situated multiagent systems such as the state of the application environment, the knowledge representation of agents, and threading. We also learned that deriving a concrete design from the reference architecture is not self-evident, it requires a lot of effort and expertise of the designer.

4.8 Summary

In this chapter, 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 started the chapter with a summary of the main functionalities of a situated multiagent system. Then we presented the reference architecture for situated multiagent systems. The reference architecture maps the functionalities of a situated multiagent system onto a system decomposition, i.e. software elements and relationships among the elements. We presented the reference architecture by means of four 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. Appendix A provides a detailed formal specification of the various 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. To demonstrate the feasibility of the reference architecture, we referred to a framework that implements the main modules of the architecture and that is instantiated for the Packet-World and an experimental robot application.

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.

Chapter 5

Architectural Design of an AGV Transportation System

Automatic Guided Vehicles (AGVs) are fully automated, custom made vehicles that are able to transport goods in a logistic or production environment. AGV systems can be used for distributing manufactured products to storage locations or as an inter-process system between various production machines. An AGV system receives transport requests from a warehouse managing system or machine operating software, and instructs AGVs to execute the transports. AGVs are provided with control software connected to sensors and actuators to move safely through the warehouse environment. While moving, the vehicles follow specific paths in the warehouse by means of a navigation system which uses stationary beacons in the work area (e.g., laser reflectors on walls or magnet strips in the floor). To enable the AGV software to communicate with software systems on other machines, the mobile vehicles are equipped with infrastructure for wireless communication.

AGV transportation systems have to deal with dynamic and changing operating conditions. The stream of transports that enter the transportation system is typically irregular and unpredictable, AGVs can leave and re-enter the system for maintenance, production machines may have variable waiting times, etc. All kinds of disturbances can occur, particular areas in the warehouse may temporarily be closed for maintenance services, supply of goods can be delayed, loads can block paths, AGVs can fail, etc. Despite these challenging operating conditions, the system is expected to operate efficiently and robustly.

Traditionally, AGVs in a factory are directly controlled by a central server. AGVs have little autonomy: the server plans the schedule for the system as a whole, dispatches commands to the AGVs and continually polls their status. This centralized approach has successfully been deployed in numerous practical instal-

lations. The centralized server architecture has two main benefits. Since the server is a central configuration point, the control software can easily be customized to the needs of a particular project. This allows for specific per-project optimizations. A second benefit is that the system behavior is deterministic and predictable.

In a joint R&D project (Egemin Modular Control Concept project, EMC^2 [7]), the DistriNet research group and Egemin have developed an innovative version of the AGVs control system. Egemin is a Belgian manufacturer of automated logistic service systems for warehouses. Egemin builds AGVs, and deploys and maintains complete installations of multiple AGVs (and other logistic machines) for their clients, see Fig. 5.1. The goal of the project was to investigate the feasibility of

Figure 5.1: An AGV at work in a cheese factory

a *decentralized* control system to cope with new and future system requirements such as flexibility and openness. Therefore, we have applied a situated multiagent system architecture. Instead of having one computer system that is in charge of numerous complex tasks such as task assignment, routing, collision avoidance, and deadlock avoidance, in the new architecture the AGVs are provided with a considerable amount of autonomy. This opens perspectives to improve flexibility and openness of the system: the AGVs can adapt themselves to the current situation in their vicinity, order assignment is dynamic, the system can deal autonomously with AGVs leaving and re-entering the system.

The AGV transportation system provided a challenging application to investigate the feasibility of applying situated multiagent systems in practice. It allowed us to apply the various mechanisms for adaptability of situated agents in a complex industrial setting. As such, the design and implementation of the AGV transportation system have considerable contributed to the development of the

reference architecture for situated multiagent systems. In this chapter, we give an overview of the software architecture of the decentralized AGV control system. Starting from system requirements, we discuss the architectural design of the application that is based on the architectural-centric software development approach described in chapter 2, and we explain how the software architecture of the AGV control system relates to the reference architecture. We zoom in on the evaluation of the software architecture and we discuss test results collected from simulations and an implemented demonstration system.

5.1 AGV Transportation System

In this section, we introduce the AGV application. We give an overview of the functionalities of the system, and we discuss the main quality requirements. System requirements are kept fairly general, independent of any particular AGV system. In section 5.6, we zoom in on specific functionalities and quality scenario's for a concrete AGV transportation application.

5.1.1 Main Functionalities

The main functionality the system has to perform is handling *transports*, i.e. moving *loads* from one place to another. Transports are generated by *client* systems, typically a business management program, but they can also be generated by machine software or service operators. A transport is composed out of multiple *jobs*: a job is a basic task that can be assigned to an AGV. For example, picking up a load is a pick job, dropping it is a drop job and moving over a specific distance is a move job. A transport typically starts with a pick job, followed by a series of move jobs and ends with a drop job.

In order to execute transports, the main functionalities the system has to perform are:

1. Transport assignment: transports are generated by client systems and have to be assigned to AGVs that can execute them.

2. Routing: AGVs must route efficiently through the layout of the warehouse when executing their transports.

3. Gathering traffic information: although the layout of the system is static, the best route for the AGVs in general is dynamic, and depends on the actual traffic conditions and forecasts in the system. Taking into account traffic dynamics enables the system to route AGVs efficiently through the warehouse.

4. Collision avoidance: obviously, AGVs must not collide. AGVs can not cross the same intersection at the same moment, however, safety measures are also necessary when AGVs pass each other on closely located paths.

5. Deadlock avoidance: since AGVs are relatively constrained in their movements (they cannot divert from their path), the system must ensure that AGVs do not find themselves in a deadlock situation.

To perform transport tasks, AGVs are equipped with a battery as energy source. AGVs have to charge their battery at the available charging stations. Depending on the application characteristics, a vehicle recharges when its available energy goes down a certain level, or the vehicle follows a pre-defined battery charge plan, or the vehicle can perform opportunity charging, i.e. the vehicle charges when it has no work to do. Finally, when an AGV is idle it can park at a free park location.

5.1.2 Quality Requirements

Stakeholders of an AGV transportation system have various quality requirements. *Performance* is a major quality requirement, customers expect that transports are handled efficiently by the transportation system. *Configurability* is important, it allows installations to be easily tailored to client-specific demands. Obviously, an automated system is expected to be *robust*, intervention of service operators is time consuming and costly.

Besides these "traditional" qualities, evolution of the market puts forward new quality requirements. Customers request for self-managing systems, i.e. systems that are able to adapt their behavior with changing circumstances autonomously. Self-management with respect to system dynamics translates to two specific quality goals: flexibility and openness.

Flexibility refers to the system's ability to exploit opportunities and anticipate possible difficulties. In the traditional centralized approach, the assignment of transports, the routing of AGVs and the control of traffic are planned by the central server. The current planning algorithm applied by Egemin is based on predefined schedules. Schedules are rules associated with AGVs and particular locations in the layout, e.g. "if an AGV has dropped a load on location x, than that AGV has to move to node y to wait for a transport assignment". A plan can be changed, however only under exceptional conditions. E.g., when an AGV becomes defective on the way to a load, the transport can be re-assigned to another AGV. A flexible control system allows an AGV that is assigned a transport and moves toward the load, to switch tasks along the way if a more interesting transport pops up. Flexibility also enables AGVs to anticipate possible difficulties. For example, when the amount of traffic is high in a certain area, AGVs should avoid that area; or when the energy level of an AGV decreases, the AGV should anticipate this and prefer a zone near to a charge station. Another desired property is that AGVs

should be able to cope with particular situations, e.g., when a truck with loads arrives at the factory, the system should be able to reorganize itself smoothly.

Openness of an AGV transportation system refers to the system's ability to deal autonomously with AGVs leaving and (re-)entering the system. Examples are an AGV that temporarily leaves the system for maintenance, and an AGV that re-enters the system after its battery is recharged. In some cases, customers expect to be able to intervene manually during execution of the system, e.g., to force an AGV to perform a particular job.

In summary, flexibility and openness are high-ranking quality requirements for today AGV transportation systems. One possibility to tackle these new quality requirements would be to adapt the central planning approach aiming to improve the flexibility and openness of the system. In the EMC^2 project however, we investigated the feasibility to apply a new decentralized architecture to cope with the new quality requirements.

5.2 Overview of the Software Architecture of the Transportation System

In this section, we give a high-level overview of the software architecture of the AGV application and we motivate the main architectural decisions. First we explain the architectural design process. Then we show how the AGV application software is integrated with external systems, and how the application software is structured and deployed on hardware. The documentation of the software architecture with the main view packets is explained in the next section.

5.2.1 Architectural Design

The general motivation to apply a situated multiagent system to the AGV transportation system was the importance of the required qualities flexibility and openness. During architectural design, we applied the various mechanisms for adaptivity for situated multiagent systems to develop the software architecture of the AGV transportation system. The insights derived from the design, the development, and the evaluation of the AGV transportation system considerably contributed to the development of the reference architecture.

For the architectural design, we used the architectural-centric software development approach described in chapter 2. Roughly spoken, the design process consisted of the following steps. First, we have mapped the system functionality onto the basic decomposition of a situated multiagent system: agents and the environment. The system consists of two types of agents, AGV agents and transport agents, that represent autonomous entities in the application. The environment consists of the deployment context extended with an application environment that

enables the agents to access resources, to exchange information, and to coordinate their behavior. Then, we have iteratively decomposed the agents and the application environment. In each decomposition step, we selected an architectural element of the software architecture and we determined the architectural drivers (i.e. the target functional and quality attribute requirements for that element). The order in which we have refined the architectural elements was essentially based on the incremental development of the application. We started with the functionality for one AGV to drive, then followed collision avoidance, next order assignment, deadlock avoidance, etc. For each decomposition, we have selected a suitable architectural pattern to refine the architectural element. Where applicable, we have used the specification of the mechanisms for adaptivity to decompose architectural elements. The decomposition ended when a suitable level of detail was reached to allow the developers to build the software.

The software architecture of the AGV transportation system includes functionality for selective perception, behavior-based action selection with roles and situated commitments, and protocol-based communication. The insights derived from applying these mechanisms in this complex application have substantially contributed to the development of the reference architecture. An important contribution for the reference architecture is related to the interaction of the application environment with the deployment context. In particular, the design of the AGV transportation system greatly improved our insights on support for dynamics in the application environment, the synchronization of state with the deployment context, and the translation of messages and influences between the application environment and the deployment context.

An important issue of the AGV application was the physical deployment of the situated multiagent system. The reference architecture abstracts from the concrete deployment of the situated multiagent system. The architectural design of the AGV application shows how distribution can be integrated with the functionality that is provided by the reference architecture. By developing appropriate middleware, we were able to separate quite well the concern of distribution and mobility from the rest of the application logic.

5.2.2 Overview of the AGV Transportation System Software

The AGV transportation system provides the control software to handle transports in a warehouse with AGVs. Fig. 5.2 shows a general overview of the software of the AGV transportation system. The software consists of three layers. AGV application is the application-specific software that accepts transport requests and instructs AGVs to handle the transports. In the traditional systems deployed by Egemin, the AGV application software consists of a central server that instructs AGVs to perform the transport requests. In the decentralized architecture, the

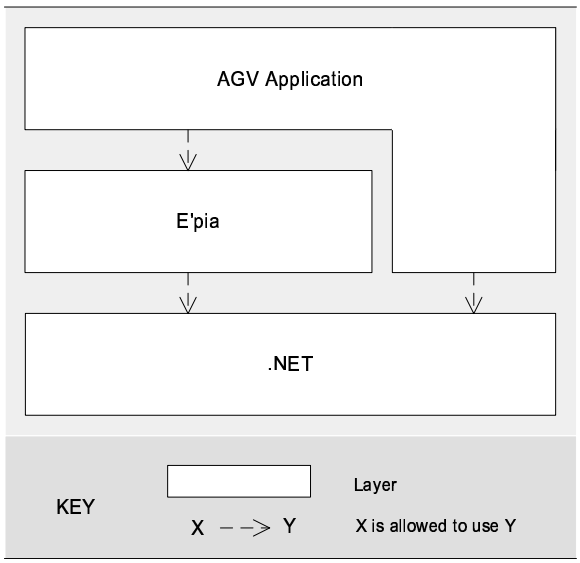


Figure 5.2: Software layers of the AGV transportation system

AGV application software consists of a situated multiagent system that handles the transport requests of the clients. The AGV application makes use of E'pia[1]. E'pia is a general purpose framework developed by Egemin that provides support for persistency, security, logging, etc. The AGV application and E'pia make use of the .NET framework [166] that provides basic support for IO, serialization, remoting, threading, etc.

Fig. 5.3 shows the context diagram of the AGV transportation system that indicates how the system interacts with external entities. Transports are requested by client systems, i.e. a warehouse management system (WMS) and a service operator. The AGV transportation system commands AGV machines to execute the transports, it monitors the status of the AGV machines, and it informs the client about the progress of the transport. The transportation system can interact with external machines and possibly command these machines to perform actions, e.g., opening a door. Besides functionality to handle transports, the software of the transportation system provides a public interface for a *monitor* to observe the status of the transportation system. The monitor is an external software system that provides a graphical user interface that allows a user to follow the activity in the transportation system. The monitor shows the actual transports in the system and the AGVs moving on the layout, and it allows a user to inspect the status of transports and AGVs.

[1]E'pia® is an acronym for Egemin Platform for Integrated Automation.

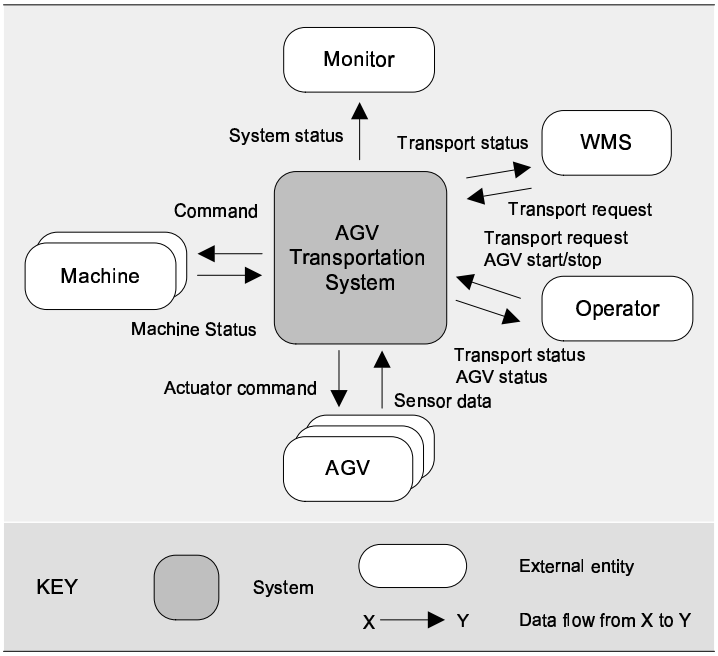


Figure 5.3: Context diagram of the AGV transportation system

5.2.3 Situated Multiagent System for the AGV Transportation System

The primary building blocks of a situated multiagent system are agents and the environment. We first introduce the two types of agents that are used in the AGV transportation system. Then, we explain the structure of the environment and we show how agents use the environment to coordinate their behavior.

5.2.3.1 AGV Agents and Transport Agents

We have introduced two types of agents: AGV agents and transport agents [222, 221]. The choice to let each AGV be controlled by an AGV agent is obvious. Transports have to be handled in negotiation with different AGVs, therefore we have introduced transport agents. An AGV agent is responsible to control its associated AGV vehicle; a transport agent represents a transport in the system and is responsible to ensure that the transport request is handled. Both types of agents share a common architectural structure, that corresponds to the agent architecture as defined in the reference architecture (section 4.3.2), yet, they have different internal structures that provide the agents with different capabilities.

AGV Agent. Each AGV in the system is controlled by an AGV agent. The AGV agent is responsible for obtaining and handling transports, and ensuring that the AGV gets maintenance on time. As such, an AGV becomes an autonomous entity that can take advantage of opportunities that occur in its vicinity, and that can enter/exit the system without interrupting the rest of the system.

Transport Agent. Each transport in the system is represented by a transport agent. A transport agent is responsible for assigning the transport to an AGV and reporting the status and completion of the transport to the client that has requested the transport. Transport agents are autonomous entities that interact with AGV agents to find suitable AGVs to execute the transports. Transport agents reside at a *transport base*, i.e. a dedicated computer located in the warehouse. Fig. 5.4 gives a schematic overview of an AGV transportation system.

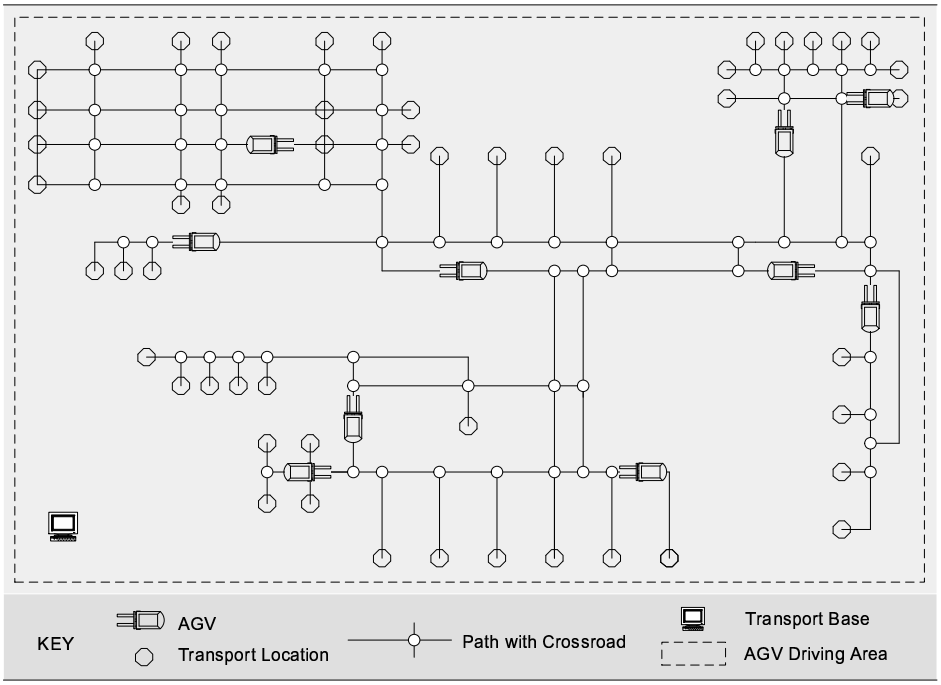


Figure 5.4: Schematic overview of an AGV transportation system

Situated agents provide a means to cope with the quality goals flexibility and openness. Particular motivations are: (1) situated agents act locally, this enables agents to better exploit opportunities and adjust their behavior with changing circumstances in the environment—this is an important property for flexibility; (2) situated agents are autonomous entities that interact with one another in their

vicinity; agents can enter and exit each others area of interaction at any time—this is an important property for openness.

5.2.3.2 Virtual Environment

To achieve the system functionality, AGV agents and transport agents have to coordinate. Agents have to coordinate for routing, for transport assignment, for collision avoidance, etc. One approach is to provide an infrastructure for communication that enables the agents to exchange messages to coordinate their behavior. Such approach however, would put the full complexity of coordination in the agents and result in complex architectures of the agents, in particular for the AGV agents. We have chosen for a solution that enables the agents to exploit the environment to coordinate their behavior [218, 215]. This approach separates responsibilities in the system and helps to manage the complexity.

The AGVs are situated in a physical environment, however, this environment is very constrained: AGVs cannot manipulate the environment, except by picking up and dropping loads. This restricts how agents can exploit their environment. We introduced a *virtual environment* that offers a medium for AGV agents and transport agents to exchange information and to coordinate their behavior. Besides, the virtual environment serves as a suitable abstraction that shields the agents from low-level issues, such as the communication of messages and the physical control of an AGV vehicle. Fig. 5.5 shows a high-level model of an AGV transportation system.

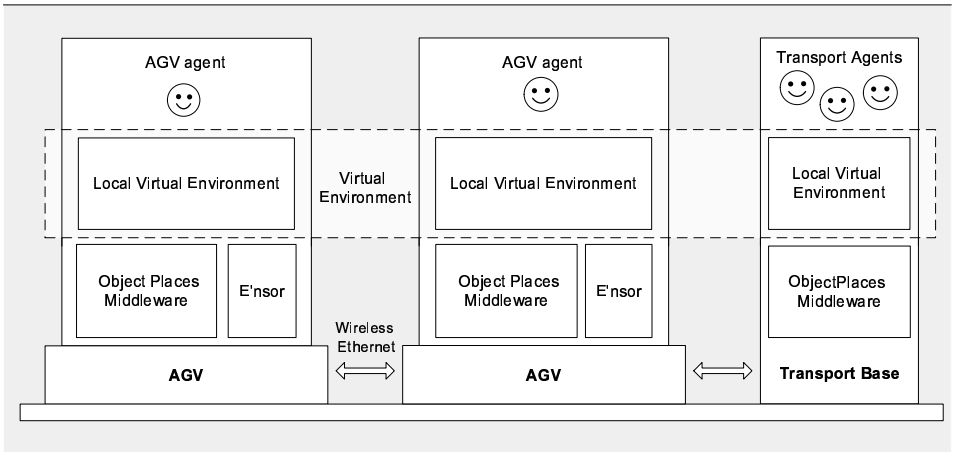


Figure 5.5: High-level model of an AGV transportation system

Since the only physical communication infrastructure available in the system is a wireless network, the virtual environment is necessarily distributed over the

AGVs and the transport base. In effect, each AGV and the transport base maintain a local virtual environment, which is a local manifestation of the virtual environment. The states of the local virtual environments are synchronized opportunistically, as the need arises. In the AGV transportation system, there is not a single software entity that corresponds to the application environment as defined in the reference architecture (section 4.3.3). Instead, the instances of the local virtual environment deployed on the nodes in the AGV system correspond to the application environment. The instances are tailored to the type of agents deployed on the nodes. For example, the local virtual environment on the AGVs provides a high-level interface that enables the AGV agent to read out the status of the AGV and send commands to the vehicle. Obviously, this functionality is not available in the local virtual environment on the transport base.

State Synchronization with ObjectPlaces. The states of local virtual environments on neighboring nodes are synchronized with each other. This synchronization is supported by the ObjectPlaces middleware [174, 176, 177]. ObjectPlaces supports the coordination of nodes in a mobile network by means of two abstractions: view and role. A view is an up to date collection of data gathered from neighboring network nodes. ObjectPlaces supports information exchange among nodes by building and maintaining views based on a declarative specification. For example, a view is used to collect the candidate AGVs that are within a range of interest for a transport agent (we discuss transport assignment in detail in section 5.4). The second abstraction, role, is a component that encapsulates the behaviour of a node in an interaction protocol. ObjectPlaces enables protocol-based interaction between nodes in a mobile network. As an example, local virtual environments use an interaction protocol for collision avoidance of AGVs (a detailed explanation follows in section 5.5). We have drawn the line with the deployment context between the local virtual environment and the ObjectPlaces middleware. In practice, a substantial part of the ObjectPlaces middleware was developed in the course of the EMC[2] project. However, a detailed discussion of ObjectPlaces is outside the scope of this dissertation.

Low-level control of AGVs with E'nsor. AGVs are equipped with low-level control software that is called E'nsor[2]. We fully reused the control software in the project. As such, E'nsor is also part of the deployment context. E'nsor provides an interface to command the AGV machine and to monitor its state. E'nsor is equiped with a map of the factory floor. This map divides the physical layout of the warehouse into logical elements: segments and nodes. Each segment and node is identified by a unique identifier. A segment typically corresponds to a physical part of a path of three to five meters. E'nsor is able to steer the AGV per segment of the warehouse layout, and the AGV can stop on every node, e.g., to change direction. E'nsor understands basic actions such as `Move(segment)` that instructs E'nsor to

[2]E'nsor® is an acronym for Egemin Navigation System On Robot.

drive the AGV over the given segment, `Pick(segment)` and `Drop(segment)` that instructs E'nsor to drive the AGV over the given segment and to pick/drop the load at the end of it, and `Charge(segment)` that instructs E'nsor to drive the AGV over a given segment to a battery charge station and start charging batteries there[3]. The physical execution of these actions, such as staying on track on a segment, turning, and the manipulation of loads are handled by E'nsor. Reading out specific sensor data, such as the current position and the battery level is also provided by E'nsor. The local virtual environment uses E'nsor to regularly poll the vehicle's current status and adjust its own state appropriately. For example, if the AGV's position has changed, the representation of the AGV position in the local virtual environment is updated.

5.2.3.3 Coordination Through the Virtual Environment

The local virtual environment offers high-level primitives to agents to act in the environment, perceive the environment, and communicate with other agents. This enables agents to share information and coordinate their behavior. We illustrate with examples how agents exploit the virtual environment as a medium for coordination.

Routing. The local virtual environment has a static layout of the paths through the warehouse. To allow an AGV agent to find its way through the warehouse efficiently, the local virtual environment provides signs on this map that the agent can use to move to a given destination. These signs can be compared to traffic signs by the road that provide directions to drivers. At each node in the map, a sign in the local virtual environment represents the cost to a given destination for each outgoing segment. The cost of the path is the sum of the costs of the segments in the path. The cost per segment is based on the average time it takes for an AGV to drive over the segment. The agent perceives the signs in its environment and uses them to determine which segment it will take next.

Transport assignment. Due to the highly dynamic nature of transport creation, the assignment of transports to AGVs is complex. To cope with the continuously changing circumstances in the environment a field-based approach is used to assign transports to AGVs. In this approach, transport agents emit fields into the local virtual environment that attract idle AGVs. To avoid multiple AGVs driving to the same transport, AGV agents emit repulsive fields. AGV agents combine received fields and follow the gradient of the combined field that guide the AGVs towards pick locations of transports. The AGV agents continuously reconsider the situation of the local virtual environment and transport assignment is delayed until the load is finally picked—which benefits the flexibility of the system.

[3]Actually, the instructions provided by the E'nsor interface are coded in a low-level digital format. The translation of actions to E'nsor instructions is handled by the local virtual environment.

Collision avoidance. AGV agents avoid collisions by coordinating with other agents through the local virtual environment. AGV agents mark the path they are going to drive in their environment using *hulls*. The hull of an AGV is the physical area the AGV occupies. A series of hulls describe the physical area an AGV occupies along a certain path. If the area is not marked by other hulls (the AGV's own hulls do not intersect with others), the AGV can move along and actually drive over the reserved path. In case of a conflict, the involved local virtual environments use the priorities of the transported loads and the vehicles to determine which AGV can move on. AGV agents monitor the local virtual environment and only instruct the AGV to move on when they are allowed. Afterwards, the AGV agents remove the markings in the environment.

These examples show that the local virtual environment serves as a flexible coordination medium: agents coordinate by putting marks in the environment, and observing marks from other agents. We discuss field-based task assignment and collision avoidance in detail in sections 5.4 and 5.5 respectively.

5.3 Documentation of the Software Architecture

We now give an overview of the software architecture documentation of the AGV transportation system. In this section, we discuss the main high-level view packages of the software architecture. In the next sections, we zoom in on two particular functionalities of the architecture: collision avoidance and transport assignment. The design of collision avoidance shows how the environment is creatively exploited in the AGV application, the design of transport assignment demonstrates how interacting agents provide a means for enhancing flexibility in the system. For the complete documentation of the software architecture of the AGV transportation system we refer to [43].

5.3.1 Deployment View of the AGV Transportation System

Fig. 5.6 gives a general overview of the AGV transportation system and shows how the system software is allocated to computer hardware. The application software consists of two types of subsystems with different responsibilities in the transportation system: transport base system and AGV control system. The relationship of the deployment view is *allocated-to* [60]. Software elements are allocated to hardware elements, e.g., a transport base system is allocated to a transport base.

5.3.1.1 Elements and their Properties

The **transport base system** provides the software to manage transports in the AGV transportation system. The transport base system handles the communication with the warehouse management system. It receives transport requests and

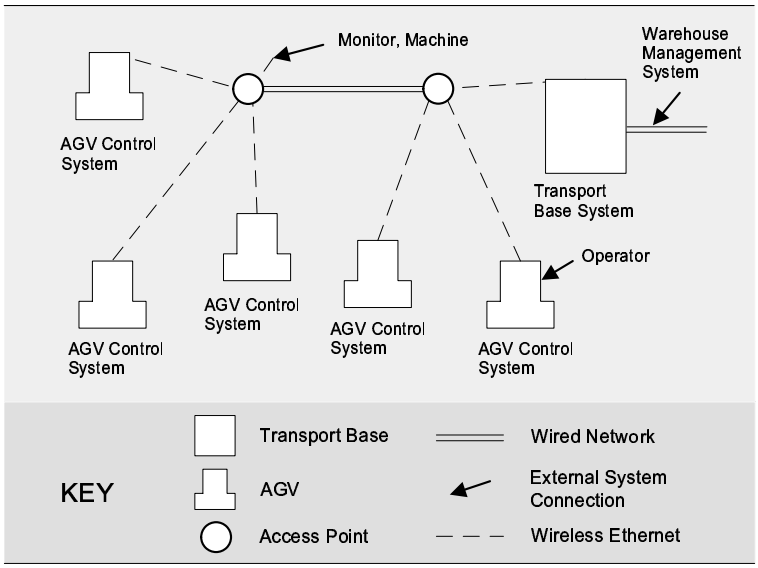


Figure 5.6: Deployment view of the AGV transportation system

assigns the transports to suitable AGVs, and it reports the status and completion of the transports to the warehouse management system. The transport base system executes on a transport base, i.e. a stationary computer. The transport base system provides a public interface that allows an external monitor system to observe the status of the AGV transportation system.

The **AGV control system** provides the control software to command an AGV machine to handle transports and to perform maintenance activities. Each AGV control system is deployed on a computer that is installed on a mobile AGV. AGV control systems provide a public interface that allows a monitor to observe the status of the AGVs, and let a service operator take over the control of the vehicle when necessary.

Communication network. All the subsystems can communicate via a wireless network. The warehouse management system interacts with the AGV transportation system via the wired network. To debug and monitor the system, AGVs and the transport base can be accessed remotely via an external monitor system.

5.3.1.2 Rationale

The main motivation for the top level decomposition of the transportation system is the separation of functionality for transport assignment (ensuring that the work

is done) from functionality for executing transports (doing the work). By providing each AGV vehicle with an AGV control system, AGVs become autonomous entities that can exploit opportunities that occur in their vicinity, and that can enter/exit the system without interrupting the rest of the system. Endowing AGVs with autonomy is a key property for flexibility and openness in the system.

The separation of functionality for transport assignment and executing transports also supports incremental development. In the initial stage of the project, we developed a basic version of the AGV control system that provided support for performing transports and avoiding collisions. For test purposes, we manually assigned transports to AGVs. In the next phase, when we extended the functionalities of AGVs and integrated automated transport assignment, the top level decomposition served as a means to assign the work to development teams.

5.3.2 Module Decomposition of the AGV Control System

Fig. 5.7 shows the primary presentation of the module uses view of the AGV control system.

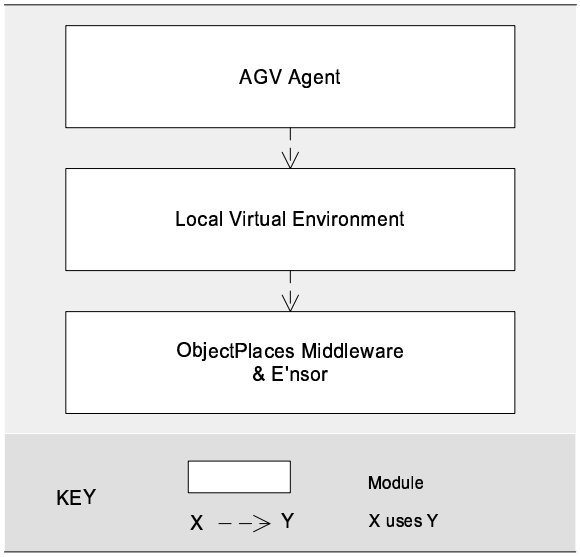


Figure 5.7: Module uses view of the AGV control system

The relation in the module uses view is *uses*. An element uses another element if the correctness of the first element depends on the correct implementation of the second element [60].

The basic structure of the AGV control system corresponds to the primary

decomposition of a situated multiagent system in the reference architecture, see Fig. 4.3 in section 4.3.1. The AGV agent is situated in the local virtual environment that corresponds to the application environment. The ObjectPlaces middleware and E'nsor are part of the deployment context.

5.3.2.1 Elements and their Properties

AGV Agent. An AGV agent is responsible for controlling an AGV vehicle. The main functionalities of an AGV agent are: (1) obtaining transport tasks; (2) handling jobs and reporting the completion of jobs; (3) avoiding collisions; (4) avoiding deadlock; (5) maintaining the AGV machine (charging battery, calibrating etc.); (6) parking when the AGV is idle.

Local Virtual Environment. Since the physical environment of AGVs is very restricted, it offers little opportunities for agents to use the environment. The local virtual environment offers a medium that the AGV agent can use to exchange information and coordinate its behavior with other agents. The local virtual environment also shields the AGV agent from low-level issues, such as the communication of messages to remote agents and the physical control of the AGV.

Particular responsibilities of the local virtual environment are: (1) representing and maintaining relevant state of the physical environment and the AGV vehicle; (2) representing additional state for coordination purposes; (3) enabling the manipulation of state; (4) synchronization of state with neighboring local virtual environments; (5) providing support to signal state changes; (6) translating the actions of the AGV agent to actuator commands of the AGV vihicle; (7) translating and dispatching messages from and to other agents.

ObjectPlaces Middleware & E'nsor. 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.

5.3.2.2 Design Rationale

The layered decomposition of the AGV control system separates responsibilities. The AGV agent is a self-managing entity that is able to flexibly adjust its behavior with changing circumstances in the environment. The local virtual environment

provides an abstraction that allows agents to interact and coordinate their behavior in a way that is not possible in the physical environment. Separation of responsibilities helps to manage complexity. An alternative for indirect coordination through the local virtual environment is an approach where the functionality to control an AGV vehicle is assigned to an AGV agent only, and where AGV agents coordinate through message exchange. Such a design however, would put the full complexity of coordination in the AGV agent, resulting in a more complex architecture.

An instance of the local virtual environment module is deployed on each node in the AGV system. As such the local virtual environment has to maintain its state with the state of other local virtual environments. Since AGV agents only interact with other agents situated in their vicinity, state has only to be synchronized between neighboring local virtual environments. The ObjectPlaces middleware takes the burden of mobility. In particular, by defining appropriate views, ObjectPlaces maintains the sets of nodes of interest for the application logic. For example, to avoid collisions, a view is defined that keeps track of all the vehicles within collision range (we discuss collision avoidance in detail in section 5.5). Whenever a vehicle enters or leaves this range, the ObjectPlaces middleware will notify the local virtual environment about the change.

5.3.3 Module Decomposition of the Transport Base System

Fig. 5.8 shows the primary presentation of the module uses view package of the transport base system.

5.3.3.1 Elements and their Properties

The **transport base manager** has a dual responsibility. First, it is responsible for the communication with client systems, it accepts transport requests and reports the status of transports to clients. Second, it is responsible for creating transport agents, i.e., for each new transport request, the transport base manager creates a new transport agent. Each transport has a priority that depends on the kind of transport. The priority of a transport typically increases with the pending time since its creation.

A **transport agent** represents a transport in the system and is responsible for: (1) assigning the transport to an AGV; (2) maintaining the state of the transport; and (3) reporting state changes of the transport to clients via the transport base manager. Physically, a transport agent is deployed on the transport base. Logically, however, the transport agent is located at a particular location in the virtual environment. For example, in section 5.4, we will discuss a field-based approach for transport assignment in which a transport agent emits a field in the environment from the location of the load of the transport to attract idle AGVs.

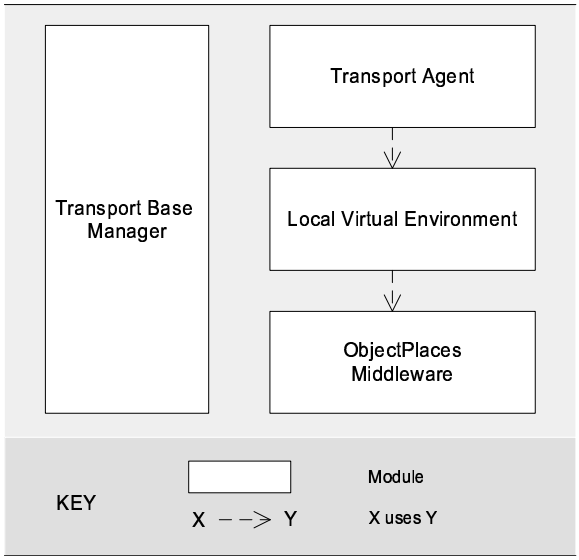


Figure 5.8: Module uses view of the transport base system

Local Virtual Environment and ObjectPlaces Middleware. The local virtual environment of the transport base system enables transport agents to coordinate with AGV agents to find suitable AGVs to execute the transports. Each transport agent has a limited view on the local virtual environment, i.e. each transport agent can only interact with the AGV agents within a particular range from its position. Yet, the range of interaction may dynamically change. In section 5.4, we explain how the range of interaction of a transport agent dynamically extends when the agent does not find a suitable AGV to execute the transport. Limiting the scope of interaction is important to keep the processing of data under control.

Contrary to the AGV agents, the transport agents in the system share one local virtual environment. Still, the state of the local virtual environment has to be synchronized with the state of local virtual environments of AGV control systems, e.g., to maintain the positions of the AGVs in the local virtual environment of the transport base and the locations of new transports in the local virtual environments of AGVs. Since transport agents can access the local virtual environment concurrently, support for concurrent access is needed.

Particular responsibilities of the local virtual environment are: (1) representing relevant state of the physical environment; (2) representing additional state for coordination purposes; (3) synchronization of state with other local virtual environments (in particular, maintaining the position of the AGVs in the system); (4) providing support to signal state changes; (5) providing support for concurrent access; (6) translating and dispatching messages from and to AGV agents.

Obviously, the responsibilities of the local virtual environment on AGVs that are related to the AGV vehicle (representing state of the AGV, translating actuator commands, etc.) are not applicable for the local virtual environment of the transport base system. The responsibilities of the ObjectPlaces middleware are similar as for the AGV control system, see section 5.3.2.

5.3.3.2 Design Rationale

The transport base is in charge of handling the transports requested by its associated clients. The transport base manager serves as an intermediary between the clients and the AGV transportation system. Apart from the transport base manager, the software architecture of the transport base is similar to the architecture of the AGV control system. Transport agents are situated in the local virtual environment that enables the agents to find suitable AGVs to perform the transport orders. The ObjectPlaces middleware that is part of the deployment context enables communication with the software systems on other nodes. The motivations for the decomposition of the transport base system are the same as for the AGV control system, see section 5.3.2.

5.3.4 Collaborating Components View of the AGV Agent

We now zoom in on the software architecture of agents. We focus on the AGV agent. Fig. 5.9 shows a collaborating components view of the AGV agent.

The general structure of the AGV agent corresponds to the structure of an agent in the reference architecture, see Fig. 4.4 in section 4.3.2. The current knowledge repository corresponds to the knowledge repository of an agent in the reference architecture, see Fig. 4.7 in section 4.4.1. The various components and the repository are further refined according to the specific functionalities of an AGV agent.

5.3.4.1 Elements and their Properties

The **Current Knowledge** repository contains state that the agent uses for decision making and communication. Current knowledge consists of static state and dynamic state. An example of static state is the value of LockAheadDistance (this parameter determines the length of the path AGVs have to reserve when they move on to avoid collisions; we elaborate on path locking in section 5.5). Examples of dynamic state are state collected from the observation of the environment such as the positions of neighboring AGVs, state of commitments related to collaborations with other agents, and runtime state related to the agent itself such as the battery status of the AGV. The current knowledge repository provides support for synchronized access. It offers a shared interface to the communication and decision making components that can concurrently read and write state. The

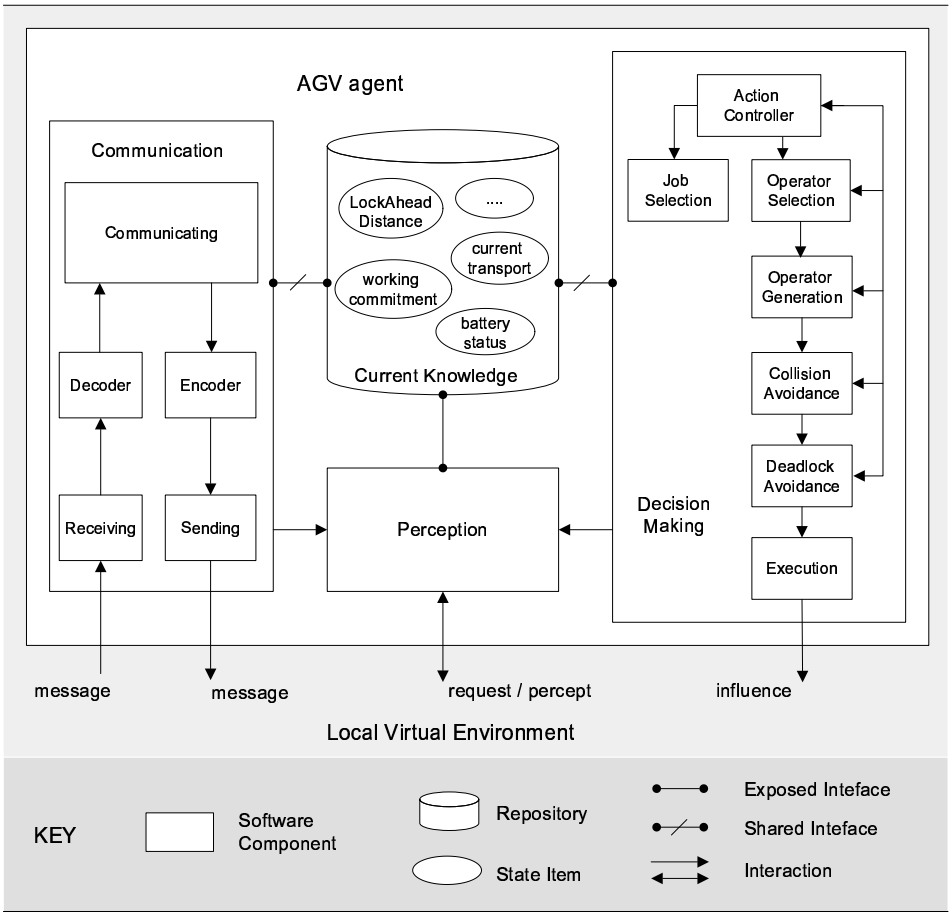


Figure 5.9: Collaborating components view of the AGV agent

perception component is connected to a separate interface to update the agents dynamic state according to the representations derived from observing the local virtual environment.

Perception enables the AGV agent to sense the local virtual environment according to the perception requests of communication and decision making, and to update the agent's current knowledge accordingly. AGV agents use different foci to sense the state of the local virtual environment that represents state in the physical environment (e.g., the positions of neighboring AGVs) and state that relates to virtual representations (e.g., fields that are used for transport assignment, see section 5.4).

The **Communication** component handles the agents communicative interactions with other agents. The main functionality of communication in the AGV transportation is handling messages to assign transports. The communicating component encapsulates the behavior of the communication component, including the protocols and the communication language used by the AGV agent. The protocols are specified as state charts, we discuss an example of a communication protocol in section 5.4.

The **Decision Making** component handles the agent's actions. In the reference architecture, action selection of a situated agent is based on a behavior-based action selection mechanism extended with roles and situated commitments (see section 4.5.2). Due to the complexity of decision making of the AGV agent, we have modelled the decision making component as a hybrid architecture that combines a blackboard pattern with sequential processing. This architecture combines complex interpretation of data with decision making at subsequent levels of abstraction. The current knowledge repository serves as blackboard, while the action controller coordinates the selection of a suitable operator. After job selection, the operator selection component selects an operator at a fairly high level (`move`, `pick`, `drop`, etc.). The operator selection component is designed as a free-flow tree extended with the notions of role and situated commitment. The main roles of the AGV agent are `work`, `charge`, and `park`. The main situated commitments are `working commitment` and `charging commitment`. A general explanation of a free-flow tree that is similar to the tree of the AGV agent is given in chapter 3, see section 3.4.3.2. The operator generation component transforms the selected operator into a concrete operator (e.g., `move(segment x)`). Collision avoidance and deadlock avoidance are responsible to lock the trajectory associated with the selected operator. As soon as the trajectory is locked, the selected operator is passed to the execution component that converts the operator to an influence that is invoked in the local virtual environment. If during the subsequent phases of decision making the selected operator can not be executed, feedback is sent to the action controller that will inform the appropriate component to revise its decision. For example, if the operator generation component has selected an operator `move(segment x)` and the collision avoidance module detects that there is a long waiting time for this segment, it informs the action controller that in turn instructs the action generation component to consider an alternative route.

5.3.4.2 Design Rationale

The current knowledge repository enables the data accessors to share state and to communicate indirectly. Communication and decision making act in parallel, each component in its own pace, supporting flexibility. Communication in the AGV application happens at a much higher pace than action selection. This difference in execution speed is exploited to continuously reconsider transport assignment in

the period between an AGV starts moving towards a load and the moment when the AGV picks the load (a detailed explanation follows in section 5.4).

Since the representation of the internal state of AGV agents and the observable state of the local virtual environment are similar (examples are the status of the battery and the positions of AGVs), we were able to use the same data types to represent both types of state. As such, no descriptions were needed to interpret representations resulting from sensing the local virtual environment. This resulted in a simple design of the perception module.

For an efficient design of the communication module, we have defined a specific communication language and an ontology that is tailored to the needs of the AGV transportation system. Since only a limited set of performatives were needed, and inter-operability was not an issue in the project, reusing an existing standard library (such as provided by e.g., Jade [38]) would have caused too much overhead.

In the initial phase of the project, we used a free-flow tree for decision making. However, with the integration of collision avoidance and deadlock avoidance, it became clear that the complexity of the tree was no longer manageable. Therefore we decided to apply an architecture that allows incremental decision making. At the top level, a free-flow tree is still used to select an operator at a high-level of abstraction; this preserves the advantage of adaptive action selection with a free-flow tree. At the following levels, the selected operator is further refined taking into account collision avoidance and deadlock avoidance. Each component in the chain is able to send feedback to the action controller to revise the decision. This feedback loop further helps to improve flexible decision making.

5.3.5 Collaborating Components View of the Local Virtual Environment

We now zoom in on the software architecture of the local virtual environment. We focus on the local virtual environment of the AGVs. Fig. 5.10 shows the collaborating components view of the local virtual environment.

The general structure of the local virtual environment is related to the structure of the application environment in the reference architecture as follows. The state repository corresponds to the state repository in the reference architecture, see Fig. 4.8 in section 4.4.2. The state elements are specific to the local virtual environment of an AGV control system. The perception manager provides the functionality for selective perception of the environment, similar to the representation generator in the reference architecture, see section 4.3.3. Contrary to the representation generator, the perception manager interacts only with the state repository; the functionality of the observation & data processing module in the reference architecture is absent in the local virtual environment. The action manager integrates the functionality of the interaction module of the application environment and the translation of influences in the deployment context. The

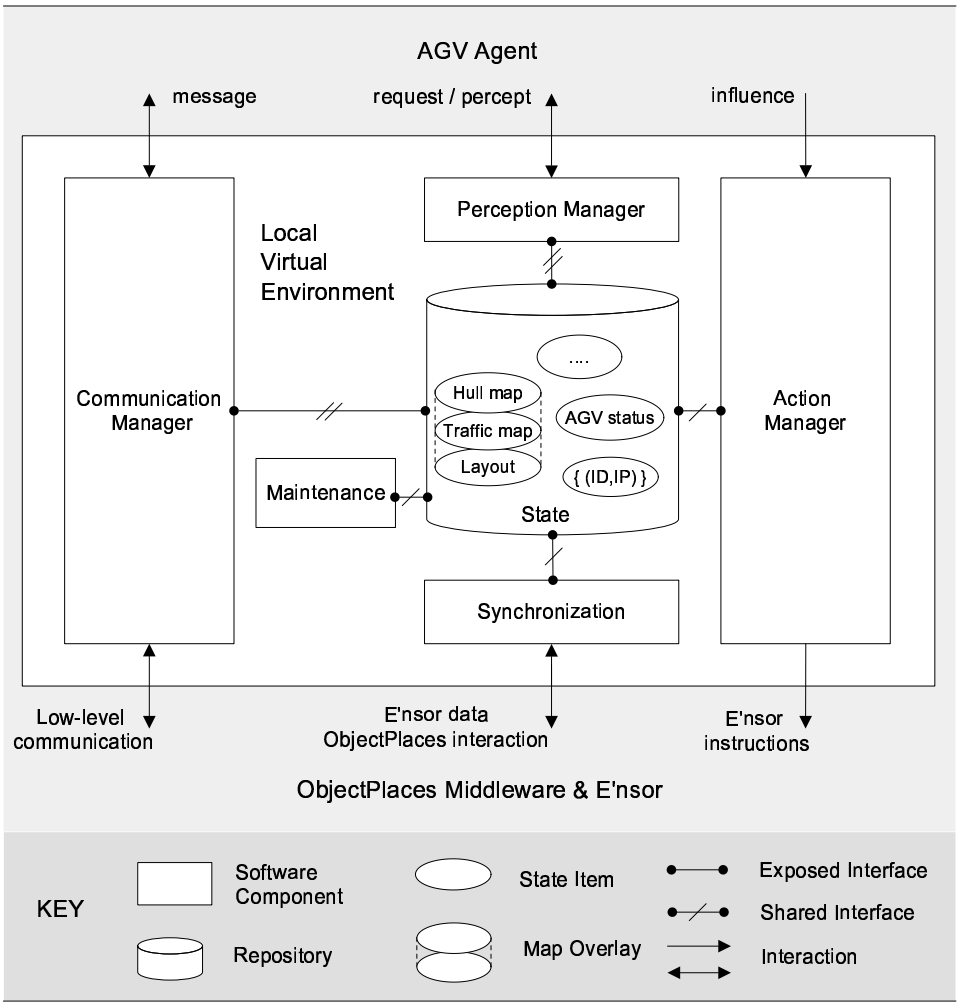


Figure 5.10: Collaborating components view of the local virtual environment of AGVs

communication manager integrates the functionality of the communication service of the application environment and the bidirectional translation of messages with the deployment context. Finally, the laws for perception, action, and communication, are not explicitly modelled in the local virtual environment, but integrated in the applicable components.

5.3.5.1 Elements and their Properties

State. Since the virtual environment is necessarily distributed over the AGVs and the transport base, each local virtual environment is responsible to keep its state synchronized with other local virtual environments. The state of the local virtual environment is divided into three categories:

1. Static state: this is state that does not change over time. Examples are the layout of the factory floor, which is needed for the AGV agent to navigate, and `(AGV id, IP number)` tuples used for communication. Static state must never be exchanged between local virtual environments since it is common knowledge and never changes.

2. Observable state: this is state that can be changed in one local virtual environment, while other local virtual environments can only observe the state. An AGV obtains this kind of state from its sensors directly. An example is an AGV's position. Local virtual environments are able to observe another AGV's position, but only the local virtual environment on the AGV itself is able to read it from its sensor, and change the representation of the position in the local virtual environment. No conflict arises between two local virtual environments concerning the update of observable state.

3. Shared state: this is state that can be modified in two local virtual environments concurrently. An example is a hull map with marks that indicate where AGVs intend to drive—we explain the use of hull maps in detail when we discuss collision avoidance in section 5.5. When the local virtual environments on different machines synchronize, the local virtual environments must generate a consistent and up-to-date state in both local virtual environments.

Perception Manager handles perception in the local virtual environment. The perception manager's task is straightforward: when the agent requests a percept, for example the current positions of neighboring AGVs, the perception manager queries the necessary information from the state repository of the local virtual environment and returns the percept to the agent. Perception is subject to laws that restrict agents perception of the virtual environment. For example, when an agent senses the hulls for collision avoidance of neighboring AGVs, only the hulls within collision range are returned to the AGV agent (for details see section 5.5).

Action Manager handles agents' influences. AGV agents can perform two kinds of influences. One kind of influences are commands to the AGV, for example moving over a segment and picking up a load. These influences are handled fairly easily by translating them and passing them to the E'nsor control software. Obviously, transport agents do not perform this kind of influences. A second kind of influences attempt to manipulate the state of the local virtual environment. Putting marks

in the local virtual environment is an example. An influence that changes the state of the local virtual environment may in turn trigger state changes of neighboring local virtual environments (see Synchronization below). Influences are subject to laws, e.g., when an AGV projects a hull in the local virtual environment, this latter determines when an AGV acquires the right to move on.

Communication Manager is responsible for exchanging messages between agents. Agents can communicate with other agents through the virtual environment. A typical example is an AGV agent that communicates with a transport agent to assign a transport. Another example is an AGV agent that requests the AGV agent of a waiting AGV to move out of the way. The communication manager translates the high-level messages to low-level communication instructions that can be sent through the network and vise versa (resolving agent names to IP numbers, etc.). Communication is subject to laws, an example is the restriction of communication range for messages used in the field-based approach for transport assignment, see section 5.4.

Maintenance is responsible for maintaining dynamism in the local virtual environment. We give an example of such dynamism when we discuss field-based transport assignment in section 5.4.

Synchronization has a dual responsibility. It periodically polls E'nsor and updates the state of the local virtual environment accordingly. An example is the maintenance of the actual position of the AGV in the local virtual environment. Furthermore, synchronization is responsible for synchronizing state between local virtual environments of neighboring machines. To synchronize state among local virtual environments, the interaction between the synchronization component and the middleware is bidirectional. We give examples of such an update process when we discuss collision avoidance in section 5.5.

5.3.5.2 Design Rationale

Different functionalities provided by the local virtual environment are assigned to different components. This helps architects and developers to focus on specific aspects of the functionality of the local virtual environment. It also helps to accommodate change and to update one component without affecting the others.

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

Since an AGV agent continuously needs up-to-date data about the system (position of the vehicles, status of the battery, etc.), we decided to keep the representation of the relevant state of the deployment context in the local virtual environment synchronized with the actual state. Therefore, E'nsor and the Ob-

jectPlaces middleware are periodically polled to update the status of the system. As such, the state repository maintains an accurate representation of the state of the system to the AGV agent.

Laws are not explicitly modelled in the local virtual environment, but integrated in the corresponding components. This improves efficiency, since there is no need to look-up the applicable laws from a repository. On the other hand, the design becomes less organized since laws are scattered over different elements.

The general architecture of the local virtual environment of the AGV application considerably contributed to the development of the application environment of the reference architecture. However, for the reference architecture we decided to split up the functionalities of the perception, interaction, and communication managers, and to introduce an explicit representation of laws in the system. Section 4.3.3.5 explains the motivations in detail.

5.4 Transport Assignment

We now explain transport assignment in the AGV transportation system. Traditional AGV systems use so called "schedule-based transport assignment". A schedule defines a number of rules that are associated with a particular location and is only valid for that location. The rules define what a vehicle has to do when it visits the schedule's associated location. The transportation system itself determines when the schedule is triggered. In other words, the moment of triggering a schedule depends on the current situation of the system, e.g. the current position and status of the vehicles, loads, etc. Schedule-based transport assignment has two important advantages: (1) the behavior of the system is deterministic, and (2) transport assignment can precisely be tailored to the requirements of the application at hand. Unfortunately, the approach has also disadvantages. First, the approach is complex and labour-intensive. Layout engineers have to define all the rules manually. Second, the assignment of transports is statically defined. The approach lacks flexibility. To improve flexibility, dynamic scheduling is introduced. Dynamic scheduling allows reassignment of jobs when a vehicle is able to perform more opportune work. Yet, the approach remains limited since it only allows an AGV to perform a new pick job in very specific circumstances, for example, when an AGV drives to a park location or when it performs an opportunity charge action.

The decentralized architecture aims to provide an approach for transport assignment that enables AGVs to flexibly switch transport assignment when opportunities occur. In this section, we discuss a decentralized mechanism for transport assignment that is based on gradient fields [198, 199, 179, 178]. We explain how the mechanism works, we zoom in on the decision making component of AGV agents, and we explain test results obtained from simulations on a real map. Finally, we briefly discuss a protocol-based approach to assign transports to AGVs

and we compare this approach with field-based transport assignment.

5.4.1 Gradient Field Based Transport Assignment

Techniques based on *fields* have been put forward as a valuable approach for the coordination of agents in a metric space [74, 126, 47, 148, 127]. In this approach, elements in the environment produce fields, which are propagated in the environment in a certain range. At each position in the metric space these fields have a certain value, forming *potential fields*. These values are typically inversely proportional to the distance from the source of the field. Agents perceive the fields and combine them in a certain way. The behavior of an agent then simply consists of following the combined potential field uphill or downhill, that is, agents follow the direction of the *gradient* of the combined field. Whereas most of the existing work on field-based coordination of software agents is applied in simplified settings, we explain how we have applied the approach in a complex real-world domain.

The basic idea of field-based transport assignment is to let each idle AGV follow the gradient of a field that guides it toward a load that has to be transported. There are two types of fields in the system: (1) transports emit fields into the environment that attract idle AGVs; (2) to avoid multiple AGVs driving towards the same transport, AGVs emit repulsive fields. AGVs combine received fields and follow the gradient of the combined fields, that guide them towards pick locations of transports. The AGVs continuously reconsider the situation of the environment and transport assignment is delayed until the load is picked, which benefits the flexibility of the system. To explain the field-based approach for transport assignment, we use the scenario depicted in Fig. 5.11.

AGV agents and transport agents. Task assignment is achieved by the interaction between AGV agents and transport agents. Physically, transport agents execute at the transport base, but *conceptually* each transport agent resides at the pick location of the load of the associated transport. In particular, transport agents are situated in the local virtual environment of the transport base system and each transport agent occupies the position of the load of its associated transport in the local virtual environment.

Both AGV and transport agents emit fields in the local virtual environment, called *AGV fields* and *transport fields* respectively, see Fig. 5.11. Fields have a certain range and contain information about the source agent. AGV fields have a fixed range, while the range of transport fields is variable. Fields are refreshed at regular times, according to a predefined refresh rate.

AGV agents store received fields. When an AGV agent perceives fields, it stores the data contained in these fields in a *field-cache*. The field-cache consists of a number of cache-entries. Each cache entry contains the identity of the received field, the most recent data contained in that field and a *freshness*. The freshness is a measure of how up-to-date the cached data is. For example, in Fig. 5.11 the

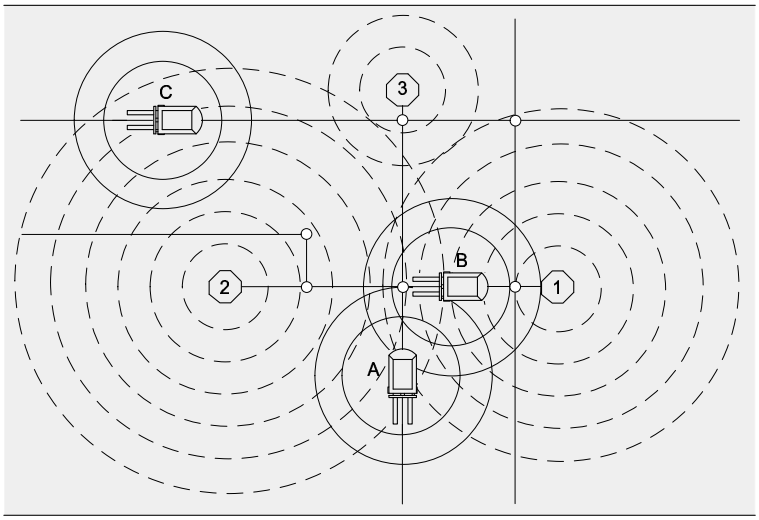


Figure 5.11: Example scenario of field-based transport assignment. Lines represent paths of the layout, small circles represent nodes at crossroads, hexagons represent transports, dotted circles represent transport fields, and full circles AGV fields.

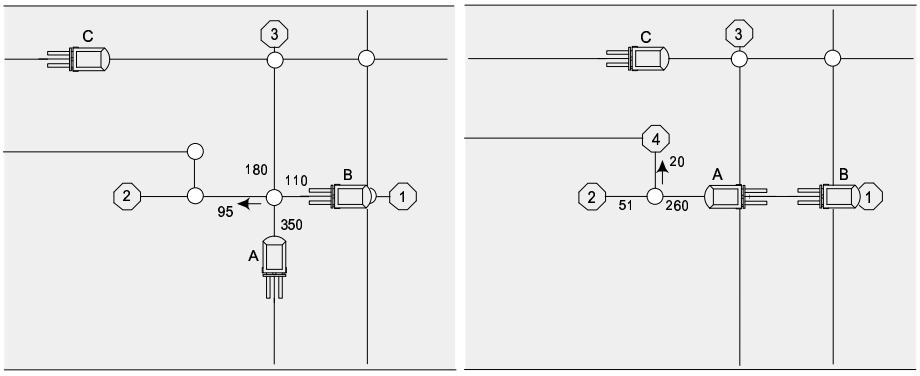


Figure 5.12: Two successive scenarios in which AGV 1 follows the gradient of the combined fields. For clarity, we have not drawn the fields.

field-cache of AGV A will consists of three entries, one for transport 1, one for transport 2, and one for AGV B.

AGV agents construct calculated-fields to decide their movement. An AGV agent constructs a *calculated-field* to decide in which direction to drive from a node. A calculated-field is a combination of the received fields, which are stored in the field-cache. The lower the freshness of a cache-entry, the lower the influence of the associated field on the calculated-field.

When an AGV is on its way from one location to another, it will lock a certain number of nodes in advance, reserving a part of the route. The reason for locking nodes is twofold: (1) it enables vehicles to avoid collisions and prevent deadlock situations; (2) it allows vehicles to keep moving, vehicles do not have to accelerate and decelerate in between two consecutive nodes. Node locking is based on a predefined distance (LockAheadDistance parameter). We elaborate on locking when we explain collision avoidance in section 5.5.

The calculated-field is therefore constructed from the last node that has been locked and contains values for each outgoing segment. An AGV agent follows the calculated-field in the direction of the smallest value. This can be considered as following the *gradient* of the calculated-field downhill.

Transport fields have an *attractive* influence on the calculated-field, which results in AGVs driving towards unassigned transports. However, it is undesirable that many AGVs drive to the same transport, since they will obstruct each other and travel superfluous distance. To remedy this, AGV fields have a *repulsive* influence.

In the left part of Fig. 5.12, AGV A constructs a calculated-field on a node (circle at the crossroad of the paths). Although transport 1 is closer, the calculated-field will guide AGV A towards transport 2. This is the result of the repulsive effect of AGV B. Notice that it would have been ineffective for AGV A to drive towards transport 1, since AGV B is closer and likely will reach the transport first.

Task assignment occurs at pick up. Task assignment is delayed until an AGV actually reaches the pick location and picks up the load. This results in a greater flexibility with respect to task assignment, in comparison with the traditional centralized approach. By delaying task assignment, the field-based approach can cope with unforseen situations like transports which suddenly popup, as illustrated in the left part of Fig. 5.12. While AGV A is driving towards transport 2, a new transport (transport 4) appears close to AGV A. Since no transport has been assigned to AGV A yet, it can drive towards the closer transport 4.

5.4.1.1 Software Architecture

We limit the discussion of the software architecture to the decision making component of the AGV agent, see Fig. 5.13. The architecture corresponds to the decision making component defined in the reference architecture extended with a number

of additional elements (see the discussion in section 5.3.4). Transport agents have a similar but more simple decision making component since these agents have only to deal with emitting fields. We discuss the various components of the decision

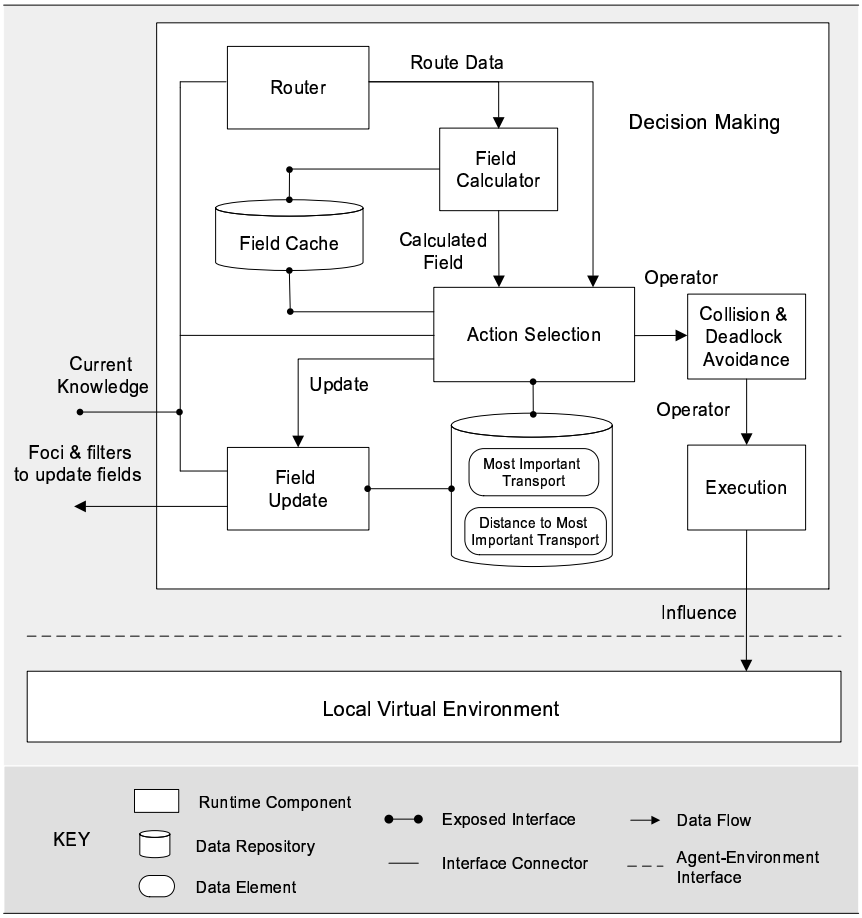


Figure 5.13: Decision making component of the AGV agent for field-based transport assignment

making component in turn. For a detailed discussion of the architecture of field-based transport assignment, we refer to [198, 179].

Field-cache: This repository stores the information of fields of other AGV agents and transport agents in cache-entries. A freshness is associated with each cache-entry, which is a measurement of how up-to-date the entry is.

Router. The router uses a map of the warehouse layout with nodes and segments

to calculate paths and distances from one node to another. For testing, we have used a static router that uses the A* algorithm [88]. However, the approach is compatible with a dynamic router that would take into account dynamic runtime information such as traffic distribution and blocked segments.

Field calculator. The field calculator constructs a calculated-field from the last selected target node by combining the received fields, which are stored in the field-cache. The higher the freshness of a cache-entry, the more influence the field associated with the cache-entry will have on the construction of the calculated-field. Thus, although still used, less importance is given to outdated information. The field calculator makes use of the router to calculate the values of the calculated-field on different positions. The gradient of the calculated-field is used as driving direction on the target node.

Field update. The field update component requests perception updates to update the fields for the AGV agent. Therefore, the field update component selects a field focus and an appropriate set and filters to sense the environment. Field update requests are periodically invoked by the action selection component.

Action selection: The action selection component continuously reconsiders the dynamic conditions in the environment and selects appropriate actions to perform the agent's tasks. Action selection passes the selected operator to the Collision & Deadlock Avoidance component that locks the path associated with the selected operator. As soon as the path is locked, the operator is passed to the Execution component that invokes an influence in the local virtual environment. Action selection maintains the agent's *current most important transport* and the *distance to the most important transport*. These values are used to achieve the repulsive influence of AGV agents. The action selection component of a transport agent maintains, among other data, the current *priority* of the transport and the *field range*. To avoid starvation of the transport, the priority of the transport grows over time. The field-range of the transport agent is a function of time and the number of interested AGV agents. The following two high-level descriptions summarize the behavior of the agents during task assignment:

```
{Action selection procedure of the AGV agent}
while idle
 do repeat with constant frequency {
  1. Sense fields and enter them in the field-cache
  2. Select operator
  3. Perform influence
}

{Action selection procedure of the transport agent}
while not assigned
 do repeat with constant frequency {
```

```
  1. Calculate priority and field-range
  2. Update status
}
```

The local virtual environment is responsible for spreading the fields. Field management is a dynamic process that takes into account the status of the agents. As soon as an AGV has picked up a load, it will inform the transport agent and execute the transport.

5.4.1.2 Test Results

To evaluate the field-based approach for transport assignment, we have performed a series of simulation tests on the map of a real layout that has been implemented by Egemin at EuroBaltic, see Fig. 5.14. The size of the physical layout is 134 m

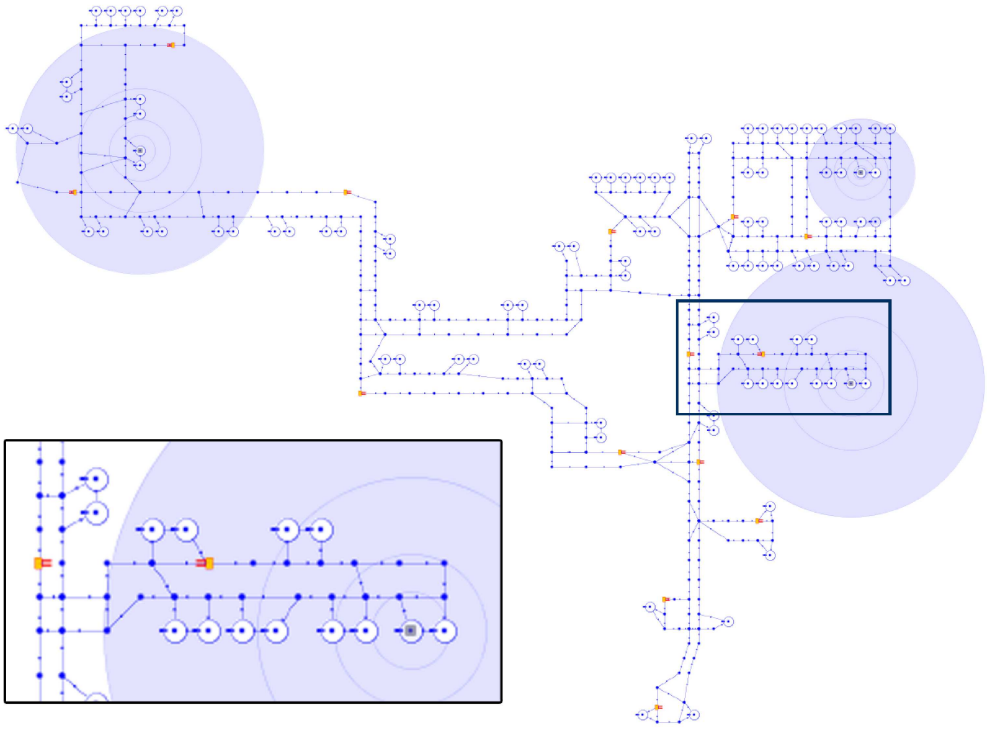


Figure 5.14: Map used for testing transport assignment

x 134 m. The map has 56 pick and 50 drop locations. The six locations at the lower side of the map are only pick locations, all other locations are pick and drop

locations. For convenience, a number of small modifications were performed to the map, e.g., all curved segments were changed to straight lines.

Transport profile and AGVs. We used a standard transport test profile that is used by Egemin for testing purposes. This profile generates 140 transports with a random pick location and a random drop location per simulation run, corresponding to the number of transports that are generated in one hour real time.

In the simulation, we used 14 AGVs just as in the real application. The average driving speed of AGVs is 0.7 m/s, while pick and drop actions take an average amount of time of 5 s.

Simulation and metrics. We used an AGV simulator for testing the field-based approach for task assignment [1]. The simulator uses a framework for time management [91, 90] to make sure the simulation results are independent of performance characteristics of the execution platform, working load of the processor, and amount of memory. Tests were run on a cluster of 40 machines: P4 2Ghz, 512MB RAM, Debian Stable 3.0.

Every simulation was run for 50000 timesteps, corresponding to approximately one hour real time, i.e. one time step represents 20 ms in real time. All test results are average values over 20 simulation runs. Performance is measured in terms of throughput and reaction time. Additionally, the amount of communication needed is measured.

Reference algorithm: Contract-Net. We used standard Contract-Net (CNET [183, 237]) as a reference algorithm to compare the field-based approach. With CNET each transport that enters the system is assigned as soon as possible to the most suitable AGV (i.e., an idle AGV for which the cost to reach the pick location is minimal). When transports can not be assigned immediately, they enter a waiting status. All waiting transports are ordered by priority, and this priority determines the order in which transports are assigned. The priority of transports grows over time, the same way as in the field-based approach.

Refreshment of fields. Maintenance of fields is achieved by periodically broadcasting the status of the fields (both position and strength). Obviously, the period between subsequent broadcasts strongly affects the amount of messages being sent. Making the period too short will produce a huge number of useless messages, but each agent in the system will have up-to-date information. On the other hand, if the broadcast period is too long, AGVs may have outdated information about the fields and probably miss some opportunities.

Fig. 5.15 shows the expected decrease in number of messages sent if the period between two broadcasts increases. The values on the X-axis express the field refresh period in time steps, e.g. a value of 20 indicates that the field status is broadcasted every 20 x 20 ms = 400 ms. With a field refresh period of 200 time steps the number of messages sent with the field-based approach is 1.7 times higher

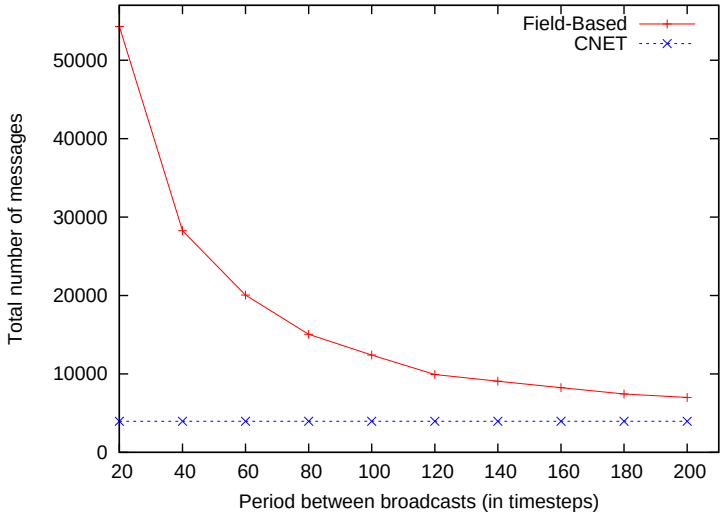


Figure 5.15: Amount of messages being sent

than with CNET. Of course what is interesting are the implications of reducing the field refresh on the performance of the system.

Performance and field refresh. Fig. 5.16 depicts the percentage of transports handled as a function of the field refresh period. It can be seen that the percentage of completed tasks fluctuates around 81,5% and slowly decreases, the difference between a refresh period of 20 and 200 is only 1% but still significantly better then the CNET approach. Fig. 5.17 illustrates that the average waiting time slowly increases with lower refresh rates, here the difference between refresh period 20 and 200 is 14%, which is still 31% better then the CNET approach.

These results clearly illustrate that communication overhead can be reduced by using a longer broadcast period, without significant performance loss. Overall, the throughput of the field-based approach is 10% higher than the throughput of CNET. The average waiting time of a transport is 39% lower, while the average number of transports waiting at each time step was around 20% lower for the field-based approach. [179] elaborates on the distance travelled by the AGVs. On average, the travelled distance of all AGVs with the field-based approach was 33% lower compared to CNET which is a result of a more optimal allocation of tasks.

Average waiting time per pick location. Although the average waiting time for transports is significantly better for the field-based approach compared to the reference algorithm, it is interesting to compare the average waiting time for trans-

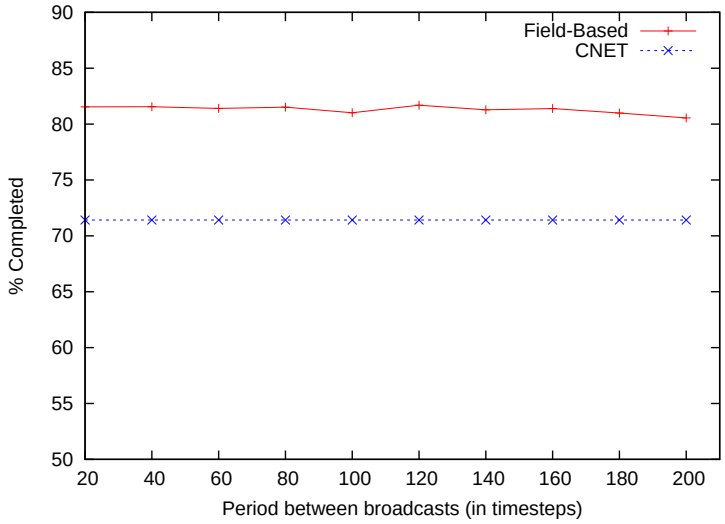


Figure 5.16: Percentage of completed transports

ports per pick location.

Fig. 5.18 shows the average waiting time for transports grouped by pick location. Clearly, the CNET reference algorithm achieves a more equal distribution. In particular, the waiting times for pick locations 1 to 5 are significantly higher for the field-based approach. This drawback can be explained as follows: because the pick locations 1 to 5 are far away from the main traffic in the warehouse, the chance an AGV will be close to the pick location is significantly lower and this decreases the chance for immediately attraction an idle AGV when a new transport pops up. Starvation is prevented since the priorities of the transports on the remote locations gradually increase when the loads are not picked. It simply takes a longer time for the field to "grow" and attract AGVs compared to the immediate assignment of an AGV in the CNET protocol. A possible remedy to this problem is to increase the strength of fields of transports on isolated locations right from the moment the transport is created.

Reflection. In the field-based approach for task assignment each idle AGV is guided toward a load of an unassigned transport by following the gradient of a field that combines attracting fields received from transports and repulsing fields received from competing AGVs. By delaying the definitive assignment of a transport until the load is finally picked, the approach achieves the required *flexibility* to exploit opportunities that may arise while AGVs search for transports in the highly dynamic environment. In addition, the field-based approach for transport

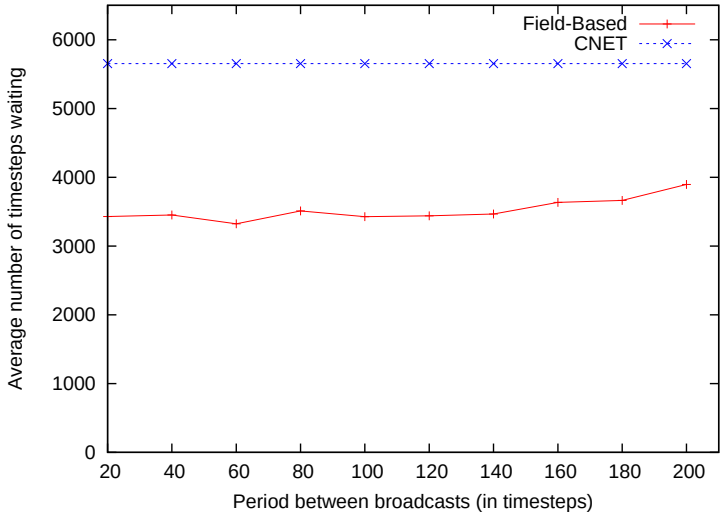


Figure 5.17: Average waiting time for transports

assignment supports *openness*. The approach easily handles AGVs that leave or re-enter the system. When for example an AGV becomes available after it has finished a transport, the AGV immediately and automatically participates in the ongoing transport assignments.

Tests show that the field-based approach outperforms standard CNET for throughput as well as for average waiting time of transports. On the other hand, the approach requires an additional amount of bandwidth, and the waiting time for isolated pick locations is higher compared to CNET. To deal with local minima, agents do not use straight distances from sources of fields to construct calculated fields, but use a router that takes into account real distances along the paths on the map. As a result, we experienced little problems with local minima.

5.4.2 Protocol-Based Transport Assignment

As an alternative to the field-based approach, we have studied and developed a protocol-based approach for transport assignment. The protocol is a dynamic version of CNET in which transport agents and idle AGV agents continuously reconsider the assignment of transports until loads are picked, aiming to exploit opportunities that arise in the environment. We call the protocol DynCNET. Whereas the coordination among transports and AGVs with gradient fields is an emergent phenomenon, the goal of the DynCNET protocol is to consider the

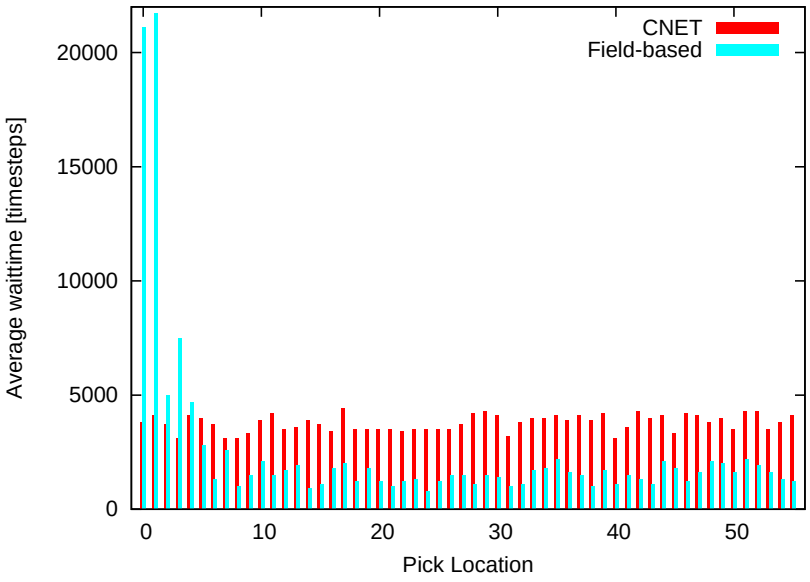


Figure 5.18: Average waiting time for transports per pick location

interaction between transport agents and AGV agents explicitly.

5.4.2.1 Protocol Description

Fig. 5.19 shows a high-level description of the DynCNET protocol for transport assignment. The protocol describes the behavior of the communication components of both AGV agent and transport agent.

In the DynCNET protocol, multiple transport agents negotiate with multiple AGV agents. First we look at the protocol from the perspective of the transport agent. Then we look at the protocol from the point of view of the AGV agent.

Protocol transport agent. When a new transport enters the system, the corresponding transport agent enters the state `Awarding` announcing the new task is available (`send(publish)`). The transport agent only contacts agents within a certain range of the pick location of the load. When no AGVs are available, the transport agent gradually increases the range to find AGVs. The set of available AGV agents is called the `InitialCandidates`. The AGV agents can bid on the transport (`send(quote)`). Once the transport agent has receive the bids of the candidate AGV agents it selects a winner (`send(win)`) and it enters the state `Assigned`. The local virtual environment—supported by the ObjectPlaces

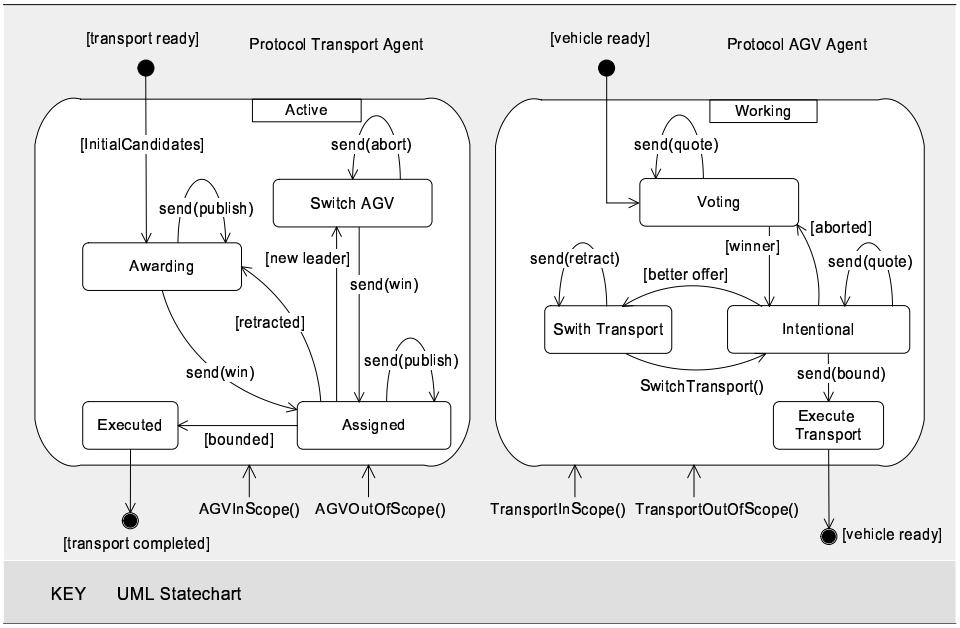


Figure 5.19: High-level description of the DynCNET protocol for transport assignment

middleware—provides a service to transport agents to inform them when AGVs come within scope to take part in the negotiation or when AGVs exit the zone of interest (`AGVInScope()` and `AGVOutOfScope()` respectively). The transport agent may switch to a new leader when a more suitable AGV becomes available (`new leader`). In such case the previous leader is informed by means of an `abort` message. Finally, when the current leader picks the load, the transport agent will be notified (the transport is `bound`) after which the transport is executed (state `Executed`).

Protocol AGV agent. Let us now look at the protocol from the point of view of the AGV agent. The agent of an idle AGV that is looking for work (`vehicle ready`) is in the `Voting` state. This agent can send quotes for announced transports. Quotes typically depend on the actual distance between the AGV and the pick locations of the loads of the announced transports. When an AGV agent is selected as winner (`winner`) it enters the state `Intentional`. The local virtual environment provides a service to AGV agents to inform them when transports enter or exit the scope of interest (`TransportInScope()` and `TransportOutOfScope()`). When the AGV agent receives an offer that is better than its current best offer,

it may switch transports. In such case, the old transport agent will receive a `retract` message. Finally, when the AGV arrives at the load of the transport for which it is the current leader, it will pick the load, inform the transport agent (`send(bound)`), and execute the transport.

The DynCNET protocol allows agents to regularly reconsider the situation in the environment and switch transport assignment when opportunities occur. Since transport agents and AGV agents have a different context, we allow both types of agents to switch transports. AGV agents can switch to a new transport that suddenly pops up, transport agents can switch to a new AGV agent that becomes available. Obviously, care must be taken that the overall behavior of the system converges and AGVs effectively perform transports. This problem was relatively easy solved by tuning a number of parameters in the protocol. Sensitive parameters are the broadcast range of the agents and the factor with which the broadcast range grows over time when no suitable agents are found have. For details about parameter tuning, see [67].

5.4.2.2 Test Results

To evaluate the DynCNET protocol, we have performed a series of simulation tests. We used the same map and test setting as for the field-based approach, see section 5.4.1.2. This allowed us to compare DynCNET with the field-based approach and CNET. For the tests, we have determined and used the most optimal parameters for the three tested protocols. Test runs were extended from 50000 to 200000 time steps, corresponding to approximately four hours real time testing. We have measured communication load, average waiting time, and the number of finished transports. In addition, we have performed a stress test in which AGVs have to handle as quickly as possible a fixed number of transports from a limited number of locations. This test simulates a peak load of transports in the system, an example is the arrival of a truck with goods that have to be distributed in a warehouse. All tests results are average values of 20 simulation runs.

Communication load. To compare the communication load, we have measured the average number of messages sent per finished transport. Fig. 5.20 shows the results of the test.

The number of messages of DynCNET is approximately the same as for the field-based approach, while the communication load of CNET is about half of the load of the dynamic protocols. However, an important difference exists between the type of messages sent. Fig. 5.21 summarizes the number of unicast and broadcast messages sent by the three protocols.

For CNET, more than 90 % of the communication are unicast messages. For DynCNET the balance unicast–broadcast messages is about 75–25, yet, for the field-based approach this balance is about 25–75, thus the opposite. This difference is an important factor for selecting appropriate communication infrastructure.

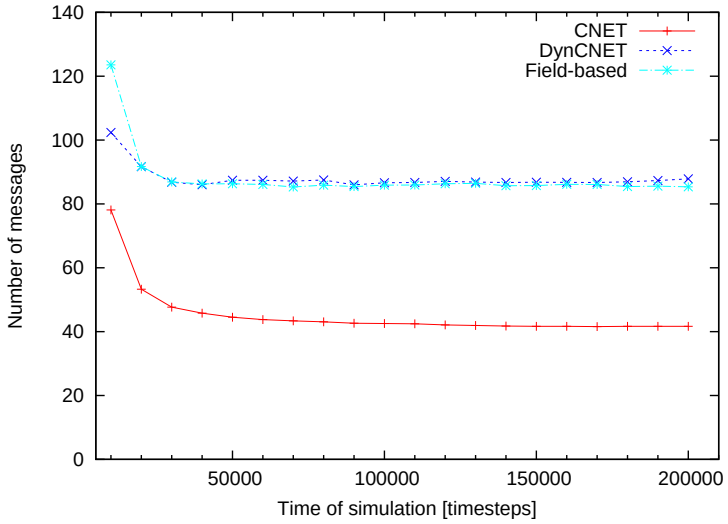
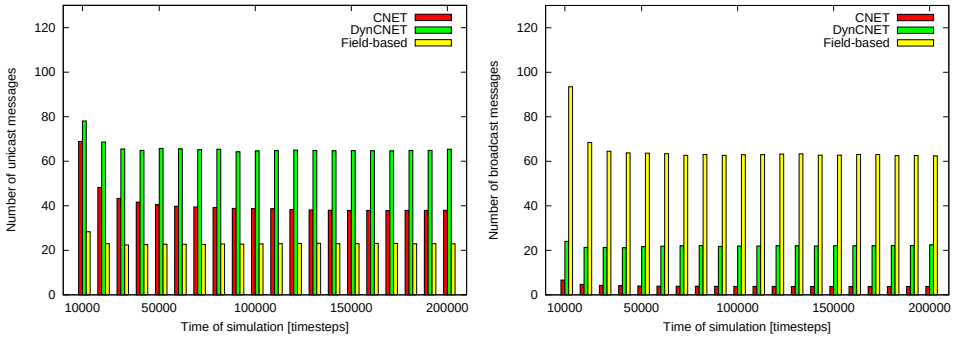


Figure 5.20: Amount of messages being sent per finished transport

Average waiting time. Fig. 5.22 shows the test results for average waiting time for transports. Average waiting time is expressed as the number of time steps a transport has to wait before an AGV picks up the load. After the transition period (around 30000 time steps), DynCNET outperforms CNET. The difference increases when time elapses. On the other hand, the field-based protocol is slightly better than DynCNET over the full test range. This came as a surprise to us, since we expected—if fine tuned well—DynCNET could perform better than the field-based approach.

Number of finished transports. Fig. 5.23 shows the number of transports finished by each of the protocols during the test run. The results confirm the measures of the average waiting time per finished transport. DynCNET handles more transports than CNET, but less than the field-based protocol. After four hours in real-time, CNET has handled 380 transports, DynCNET has handled 467 transports, and the field-based approach 515 transports. For the 467 executed transports of DynCNET, we measured an average of 414 switches of transport assignments performed by transport agents and AGV agents.

Stress test. In addition to the standard transport test profile, we have performed a stress test in which 50 transports are created at a limited number of locations in the beginning of the test. These transports have to be dropped at a particular set of destinations. The test simulates for example the arrival of a truck with loads



(a) (b)

Figure 5.21: (a) Number of unicast messages, (b) Number of broadcast messages.

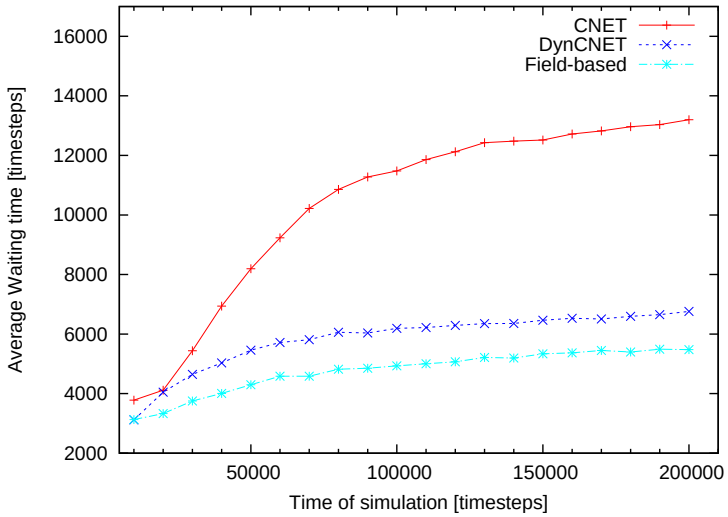


Figure 5.22: Average waiting time

that have to be distributed in a warehouse. The task of the AGVs is to bring 45 of the loads as quickly as possible to the right destinations. We limited the number of transports to 45 to avoid the effects of AGVs that hinder one another while performing the final transports. The transport test profiles for the three approaches was identical. Fig. 5.24 shows the test results.

The slopes of the curves of the field-based protocol and DynCNET protocol are

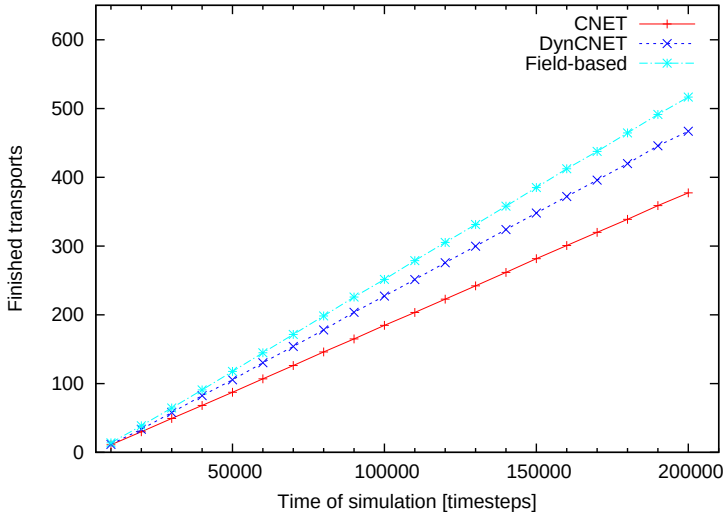


Figure 5.23: Number of finished transports

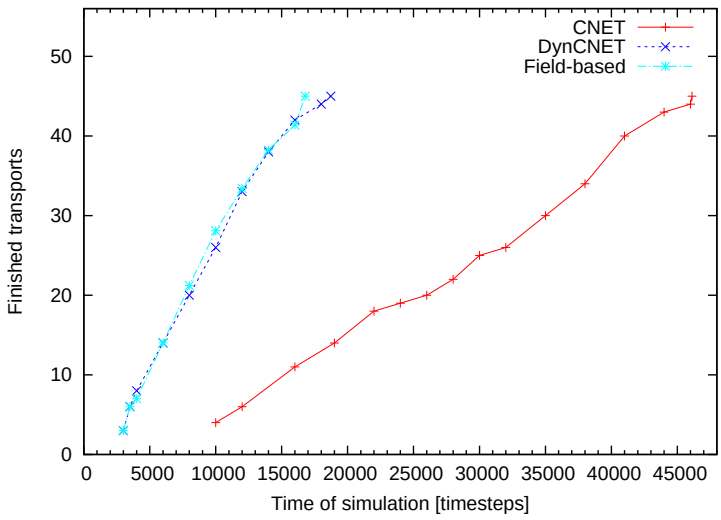


Figure 5.24: Number of finished transports in the stress test

about equal but much steeper than the curve of CNET. The results demonstrate that CNET requires about 2.5 times more time to complete the 45 transports than the dynamic protocols. The communication load for this test scenario is shown in Fig. 5.25.

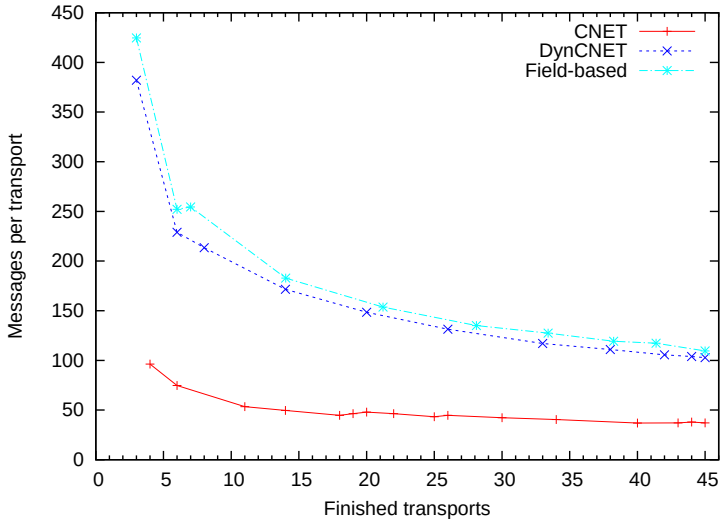


Figure 5.25: Messages per finished transports

In the stress test, the communication load of the field-based protocol is slightly higher as DynCNET during the full test. In the initial phase of the test, the dynamic protocol requires much more bandwidth as CNET (about factor four), later the difference converges to the difference we have measured for the standard test profiles (about factor two).

Variance. The tests we have discussed in the previous sections are non-deterministic. Orders are generated randomly and priorities are assigned randomly. To verify the statistic significance of the mean values of the test results we have calculated the 95 % confidence interval [197]. Fig. 5.26 shows the average waiting time per finished transport of Fig. 5.22 with a 95 % confidence interval.

The results show rather small confidence intervals. This means that the variance of the test results is small. The results of the other tests show similar values for confidence intervals. For details we refer to [67].

Reflection. DynCNET outperforms CNET on all performance measures. The cost is a doubling of required bandwidth. Yet, contrary to our expectations, DynCNET is in general not able to outperform the field-based approach. At best DynC-

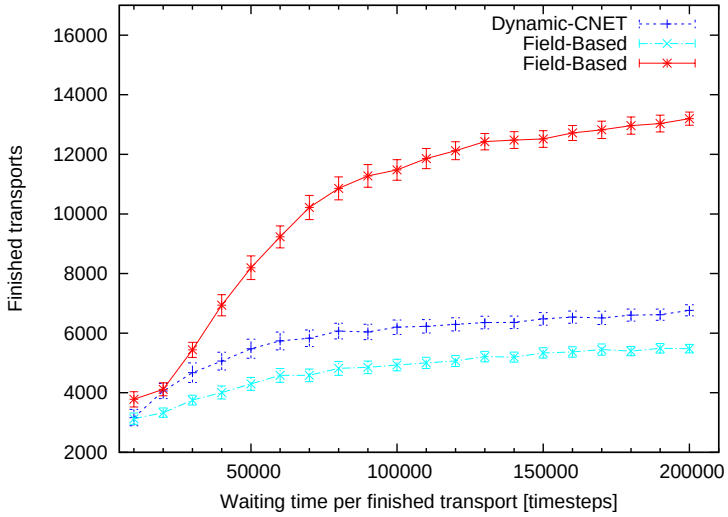


Figure 5.26: Average waiting time per finished transport with a 95 % confidence interval

NET is able to equal the performance of field-based transport assignment. With respect to flexibility, DynCNET and the field-based approach are of the same quality.

Contrary to field-based transport assignment, DynCNET defines the interactions among agents explicitly. This allows to reason about the assignment of transport in the system. It helps to better understand the overall behavior of system, and it supports fine tuning of the protocol to application specific needs. A significant difference exists in the ratio unicast–broadcast messages that are communicated in the approaches. This difference is important with respect to the applied communication infrastructure. Finally, an important distinction between DynCNET and the field-based approach concerns robustness. Whereas field-based transport assignment is robust to various failures such as a break down of an AGV and the loss of messages, DynCNET is much more sensitive. Without additional measurements, the DynCNET protocol can fail when for some reason the prescribed sequence of messages is disturbed.

5.5 Collision Avoidance

We now explain how AGVs avoid collisions. In the centralized approach, collision avoidance is realized as follows: for each AGV in the system, a series of *hulls*

are calculated along the path each AGV is going to drive. A hull represents the physical area an AGV occupies, and a *hull projection* projects a hull over a part of the path the AGV intends to drive on. When two or more hull projections overlap, AGVs are within *collision range* and all except one AGV are commanded to wait.

5.5.1 Decentralized Mechanism for Collision Avoidance

In a decentralized architecture, a central arbitrator does not exist. However, the virtual environment enables the agents to act as if they are situated in a shared environment, while the virtual environment takes on the burden of coordination. Fig. 5.27 shows a series of screenshots of a simulation run in a realistic map. In Fig. 5.27(a), two AGVs, A and B, are approaching one another. Both AGVs are projecting hulls in the environment. At this point, no conflict is detected. In Fig. 5.27(b), AGV B has projected further ahead, and is now in conflict with the hull projection of AGV A. If we assume that AGV A has already reserved the trajectory occupied by its hull, AGV A is given priority to AGV B that must wait. In 5.27(c), AGV A is taking the curve, passing AGV B. Finally, in 5.27(d), AGV A has parked at the bottom, and AGV B is moving.

We now describe the collision avoidance mechanism in more detail. First, we focus on how the agent avoids collision without being aware of the actual underlying collision avoidance protocol, then we study the work behind the scenes (i.e. the protocol) in the virtual environment.

In order to drive, the agent takes the following actions:

1. The agent determines the path it intends to follow over the layout. The agent determines how much of this path it wants to lock. This is determined by LockAheadDistance parameter that ensures that the AGV moves smoothly and stops safely[4].

2. The agent marks the path it intends to drive with a *requested hull projection*. This projection contains the agents id and a priority, that depends on the current transport the AGV is handling.

3. The agent perceives the environment to observe the result of its action.

4. The agent examines the perceived result. There are two possibilities:

   (a) The hull is marked as "locked" in the environment; it is safe to drive.

   (b) The hull is not marked as locked. This means that the agent's hull projection conflicted with that of another agent. The agent may not

[4]Besides the LockAheadDistance, the AGV also applies basic rules for deadlock avoidance such as locking a bi-directional path until the end to avoid that another AGV enters from the other direction, leading directly to a deadlock situation. Yet, we do not further elaborate on deadlock avoidance here.

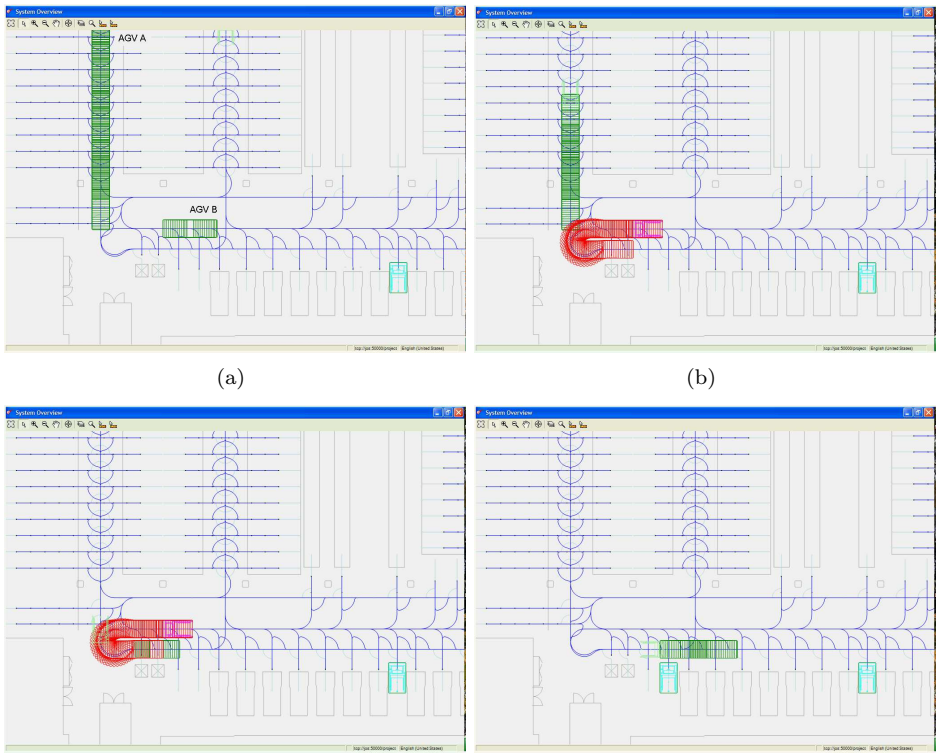


Figure 5.27: (a) Two AGVs approaching, (b) A conflict is detected, (c) One AGV passes safely, (d) The second AGV can pass as well.

pass; at this point the agent may decide to wait and look again at a later time, or remove its hull projection and take another path altogether.

The virtual environment plays an important role in this coordination approach: it must make sure that a hull projection becomes locked eventually. To this end, the local virtual environment of the AGV agent that requests a new hull projection, executes a collision avoidance protocol with local virtual environments of nearby AGVs.

It is desirable to make the set of nearby AGVs not larger than necessary, since it is not scalable to interact with every AGV in the system. On the other hand, the set must include all AGVs with which a collision is possible: safety must be guaranteed.

A solution to this problem is shown in Fig. 5.28. The local virtual environ-

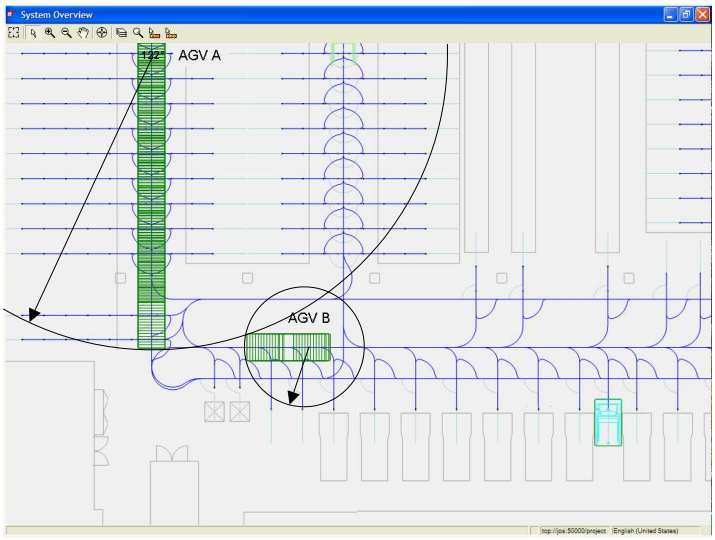


Figure 5.28: Determining nearby AGVs

ment of a *requesting AGV* will interact with the local virtual environments of other AGVs whose *hull projection circle* overlaps with the hull projection of the requesting AGV. The hull projection circle is defined by a center point, which is the position of the AGV itself, and a radius, which is equal to the distance from the AGV to the furthest point on its hull projection. If two such circles overlap, this indicates (to a first approximation) that the two AGVs might collide. We call the set of AGVs with overlapping hull projection circles the *requested AGVs*.

The local virtual environment of the requesting AGV executes the following protocol with the local virtual environment's of requested AGVs. The protocol is a variant on well-known mutual exclusion protocols based on voting.

1. The local virtual environment of the requesting AGV sends the requested hull projection to the local environments of all requested AGVs.

2. The local environments of requested AGVs check whether the projection overlaps with their hull projection. There are three possibilities for each of the requested AGVs:

   (a) No hull projections overlap. The local virtual environment of the requested AGV sends an "allowed" message to the local virtual environment of the requesting AGV.

   (b) The requesting AGV's hull projection overlaps with the requested AGV's hull projection, and the requested AGV's hull is already locked. The

local virtual environment of the requested AGV sends a "forbidden" message to the local virtual environment of the requesting AGV.

(c) The requesting AGV's hull projection overlaps with the requested AGV's hull projection, and the requested AGV's hull is not locked. Since each of the requested hulls contains a priority, the local virtual environment of the requested AGV can check which hull projection has precedence. If the hull projection of the requesting AGV has a higher priority than that of the requested AGV, the local virtual environment of the requested AGV replies "allowed"; it replies "forbidden" otherwise.

3. The local virtual environment of the requesting AGV waits for all "votes" to come in. If all local virtual environments of the requested AGVs have voted "allowed", the hull projection can be locked and the state of the local virtual environment is updated. If not, the local virtual environment of the requesting AGV waits a random amount of time and then tries again from step 1.

If at any time, the agent removes the requested hull from the virtual environment, the protocol is aborted.

The approach used for collision avoidance shows how the virtual environment serves as a flexible coordination medium, which hides much of the distribution of the system from the agents: agents coordinate by putting marks in the environment, and observing marks from other agents.

5.5.2 Software Architecture: Communicating Processes for Collision Avoidance

We now illustrate how collision avoidance is dealt with in the software architecture of the AGV transportation system. Fig. 5.29 shows the primary presentation of the communication processes view for collision avoidance.

The communicating processes view presents the basic components of the AGV control system and overlay them with the main processes and repositories involved in collision avoidance; compare the module decomposition view of the AGV transport system in Fig. 5.7 of section 5.3.2, the collaborating components of the AGV agent in Fig. 5.9 of section 5.3.4, and the collaborating components view of the local virtual environment in Fig. 5.10 of section 5.3.5. We now discuss the main architectural elements involved in collision avoidance in turn.

The **Perception Process** is part of the perception component (see section 5.3.4), 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 perception manager (see section 5.3.5), and corresponds to the Representation Generator process in the reference architecture. This process is responsible for handling perception requests, it

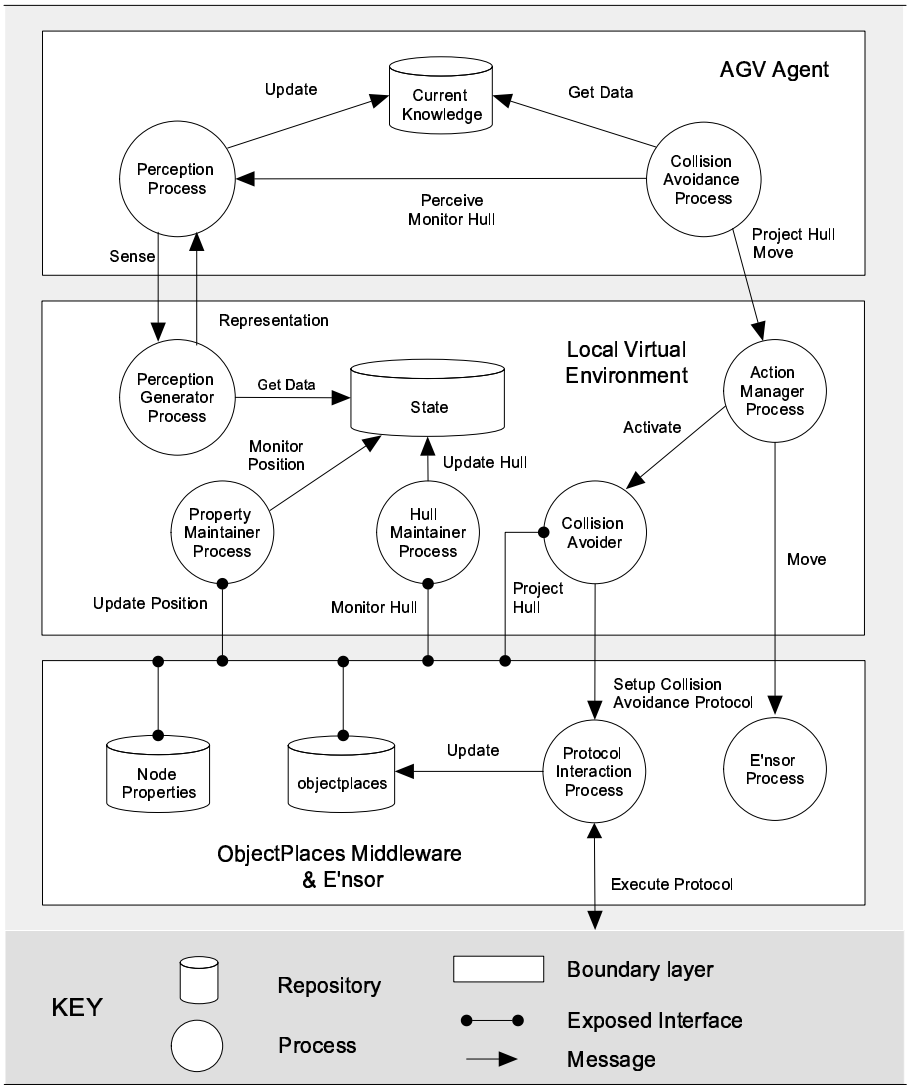


Figure 5.29: Communicating processes for collision avoidance

derives the requested data from the state repository of the local virtual environment according to the given foci of the request.

Collision Avoidance Process is part of the collision avoidance component of

decision making, see section 5.3.4. This process is an application-specific instance of the Decision Making process in the reference architecture. 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 action manager component, see section 5.3.5. This process 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 in the local virtual environment, or to the E'nsor process. For a hull projection, the action manager process passes the action to the collision avoider process.

Objectplaces repository is a repository of data objects in the ObjectPlaces middleware, see section 5.3.2. The repository 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, such as the AGV's current position. Maintenance of node properties in the repository is handled by the **Property Maintainer Process**. This process is a maintenance process of the local virtual environment, see section 5.3.5. 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 action manager (see section 5.3.4) 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** is part of the synchronization component, see section 5.3.5. This process is an application-specific instance of a Synchronization process 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.

5.6 ATAM Evaluation

For the evaluation of the software architecture of the AGV transportation system we used the Architecture Tradeoff Analysis Method (ATAM) [45, 43]. The main

goal of the ATAM is to examine whether the software architecture satisfies system requirements, in particular the quality requirements. We applied the ATAM for one concrete application, in casu a tobacco warehouse transportation system that was used as a test case in the project. In this section, we give an overview of the ATAM evaluation. We briefly introduce the tobacco warehouse application and the business goals. Then we zoom in on the utility tree and we discuss the analysis of architectural approaches for two quality attribute scenarios. The section concludes with a reflection on the ATAM experiences.

5.6.1 ATAM Workshop

In preparation to the ATAM evaluation, three stakeholders together with one of the evaluators held a four-days Quality Attribute Workshop (QAW [33]). A QAW is a facilitated method that engages stakeholders to discover the driving quality attributes of a software-intensive system. During the QAW we developed a utility tree for the tobacco warehouse transportation system.

The ATAM itself was conducted by a team of three evaluators and nine stakeholders, including a project manager, two architects, a project engineer, two developers, a service and a simulation engineer, and a representative for the customer. The workshop took one day and followed the standard ATAM phases, i.e., presentations of ATAM, business drivers, architecture and architectural approaches. Next the quality attribute utility tree was discussed with the stakeholders and two quality scenarios were analyzed in detail. The workshop initiated a number of additional activities. A number of tests were conducted to investigate the main risks that were identified during the workshop. An extra analysis of risks and tradeoffs of the software architecture was performed with a reduced number of stakeholders. Finally, the architects finished the architectural documentation, and the evaluators presented the main workshop results.

5.6.2 Tobacco Warehouse Transportation System

In the tobacco application, bins with tobacco are stored in a warehouse and AGVs have to bring the full and empty bins to different tobacco-processing machines and storage locations. The warehouse measures 75 x 55 meters with a layout of approximately 6000 nodes. The installation provides 12 AGVs that use navigation with magnets in the floor (1800 in total). There are 30 startup points for AGVs, i.e., points where AGVs can enter the system in a controlled way. AGVs use opportunity charging and a 11 Mbps wireless ethernet is available for communication. Transports are generated by a warehouse management system. The average load is 140 transports/hour, i.e., 11,5 transports/AGV. Processing machines can be in two modes: low-capacity when machines ask for bins and high-capacity mode when bins are pushed to machines. Particular opportunities for optimization are double play (a double play is a combined transport consisting of a drop action in

a predefined double play area by a specific vehicle and a pick action of a waiting load in the same area by the same vehicle), late decision for storage orders, and opportunity charging.

Important business goals for the tobacco warehouse transportation system are:

- Flexibility with respect to storage capacity, throughput, and order profiles.
- Extendibility of the layout, production lines, and the number of vehicles.
- Reliability, i.e. 99.99 % up-time, downtime may never cause production halt, and full tracing of quantities.
- Integration with ICT environment, wireless communication, security policy, and remote connectivity.

The installation is subject to a number of technical constraints, including backwards compatibility with E'pia the general purpose framework developed by Egemin that provides basic support for persistency, security, logging, etc., and compatibility with E'nsor the low-level control software deployed on AGVs. Finally, the load of the wireless network is restricted to 60 % of its full capacity.

5.6.3 Utility Tree

A utility tree provides a mechanism for architects and other stakeholders involved in a system to define and prioritize the relevant quality requirements precisely. A utility tree characterizes the driving attribute-specific requirements in a 4-level tree structure where each level provides more specific information about important quality goals with leaves specifying measurable quality attribute scenarios. Each scenario is assigned a ranking that expresses its priority relatively to the other scenarios. Prioritizing takes place in two dimensions. The first mark (High, Medium or Low) of each tuple refers to the importance of the scenario to the success of the system, the second to the difficulty to achieve the scenario.

During the QAW, 11 different qualities and 34 concrete quality attribute scenarios were specified for the tobacco warehouse transportation system. Fig. 5.30 shows an excerpt of the utility tree of the AGV system. At the ATAM workshop, minor changes were applied to the utility tree based on the discussion with the extended group of stakeholders.

5.6.4 Analysis of Architectural Approaches

Based upon the high-priority quality goals, the architectural approaches that address those factors were elicited and analyzed. During this step in the ATAM, a number of architectural risks (i.e. problematic architectural decisions), sensitivity points (i.e. architectural decision that involve architectural elements that are critical for achieving the quality attributes), and tradeoff points (i.e. architectural

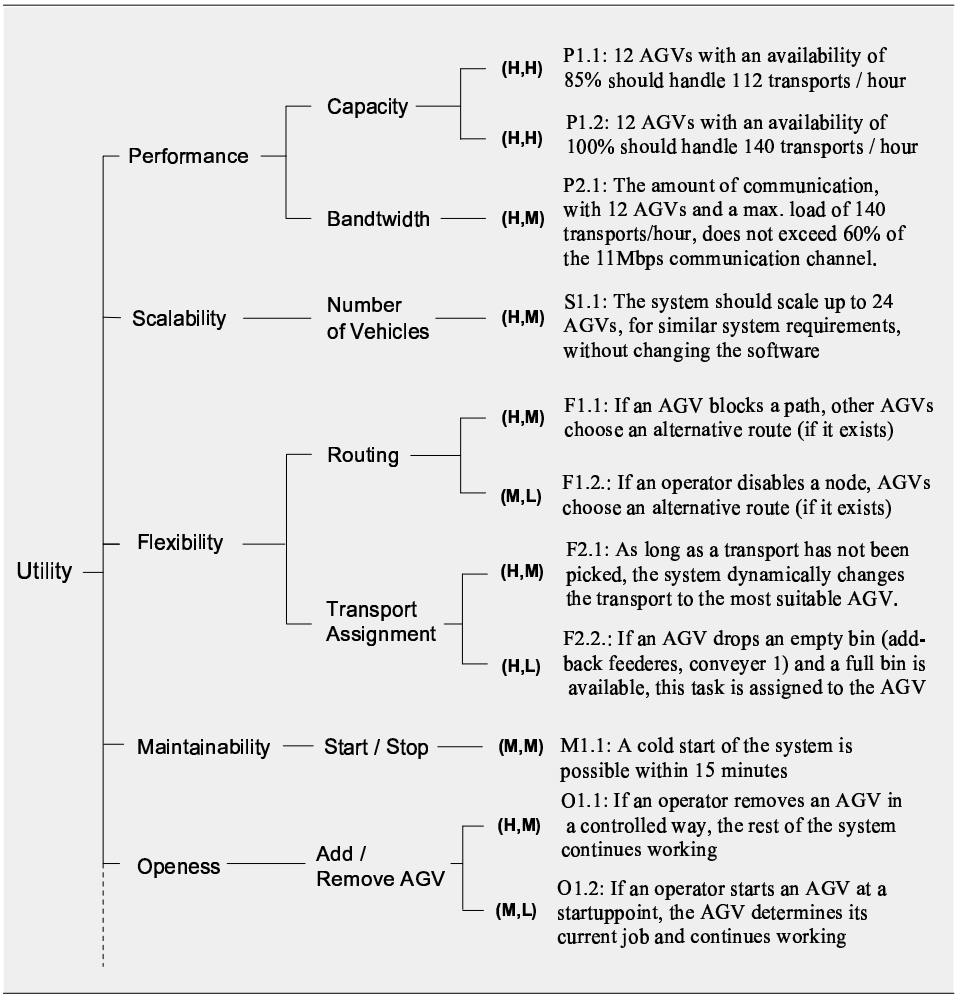


Figure 5.30: Excerpt of the utility tree for the tobacco warehouse transportation system

decisions that affect more than one attribute) of the software architecture were identified. The group of stakeholders discussed two particular quality attribute scenarios: one scenario concerning flexibility (transport assignment) and another scenario concerning performance (bandwidth usage). We give an overview of the results of the analysis of the two scenarios.

5.6.4.1 Architectural Analysis of Flexibility

Fig. 5.31 shows an overview of the analysis of architectural decisions for the main quality attribute scenario of flexibility. The table shows the main architectural

Analysis of Architectural Approach				
Scenario #: F2.1	As long as a transport has not been picked up, the system dynamically changes that transport's assignment to the most suitable AGV.			
Attributes	Flexibility			
Environment	Normal operation			
Stimulus	A transport has not been picked up and the transport's assignment can be improved.			
Response	The system dynamically changes the assignment of the transport to the most suitable AGV.			
Architectural decisions		**Sensitivity**	**Tradeoff**	**Risk**
AD 1 Negotiating agents			T1	
AD 2 Locality		S1		
AD 3 DynCNET protocol for transport assignment				R1

Figure 5.31: Analysis architectural approaches with respect to flexibility

decisions (AD) that achieve the quality attribute scenario, and specifies sensitivity points, tradeoffs, and risks associated with the architectural decisions. We give a brief explanation of the various architectural decisions:

AD 1 An agent is associated with each AGV and each transport in the system. To assign transports, multiple AGV agents negotiate with multiple transport agents. Agents continuously reconsider the changing situation, until a load is picked. The continuous reconsideration of transport assignments improves the flexibility of the system. However, it also implies a significant increase of communication. This was registered as tradeoff T1.

AD 2 For their decision making, agents take only into account local information in the environment. The most suitable range varies per type of information, and may even vary over time for one particular type of information, e.g. candidate transports, vehicles to avoid collisions, etc. The determination of

this range for various functionalities is a sensitivity point. This sensitivity point was denoted as S1.

AD 3 The dynamic Contract-Net transport assignment protocol is documented at a high-level of abstraction. At the time of the ATAM, several important decisions were not taken yet. The difficulty of parameter tuning to ensure convergence and optimal behavior was unclear. This lack of clearness was registered as risk R1.

5.6.4.2 Architectural Analysis of Bandwidth Usage

Fig. 5.32 shows an overview of the analysis of architectural decisions for the main quality attribute scenario of bandwidth usage.

We give a brief explanation of the various architectural decisions:

AD 1 The AGV transportation system software is built on top of the .NET framework. This choice was a business constraint but also an evident choice since the E'pia library that is used for logging, persistence, security, etc., also uses .NET. The overhead induced by the choice for the point-to-point communication approach of .NET remoting was registered as a sensitivity point S2.

AD 2 Each AGV vehicle is controlled by an agent that is physically deployed on the machine. This decentralized approach induces a risk with respect to the required bandwidth for inter-agent communication. This was recorded as risk R2. An AGV agent can flexibly adapt its behavior to dynamics in the environment. AGVs controlled by autonomous agents can enter/leave the system without interrupting the rest of the system. However, flexibility and openness comes with a communication cost. This tradeoff was noted as T2.

AD 3 The dynamic Contract-Net protocol for transport assignment enables flexible assignment of transports among AGVs. Yet, the continuous reconsideration of transport assignment implies a communication cost. This tradeoff was denoted as T3.

AD 4 AGV agents use a two phase deadlock prevention mechanism. AGV agents first apply static rules to avoid deadlock, e.g. agents lock unidirectional paths over their full length. These rules however, do not exclude possible deadlock situations completely. If an agent detects a deadlock, it contacts the other involved agents to resolve the problem. Yet, the implications of the deadlock mechanism on the communication overhead are at the time of the ATAM not fully understood. This lack of insight was denoted as risk R3.

AD 5 The ObjectPlaces middleware uses unicast communication. However, some messages have to be transmitted to several agents, causing overhead. Support for multicast is possible, yet, this implies that the basic support of .NET

Analysis of Architectural Approach				
Scenario #: P2.1	The amount of communication, with maximal 12 AGVs and a maximal load of 140 transporta per hour, does not exceed 60 % of the bandwidth of the 11 Mbps communication channel.			
Attributes	Performance			
Environment	All operation modes with maximal 12 AGVs and a maximal load of 140 transports per hour.			
Stimulus	Communication among subsystems.			
Response	Communication load should not exceed 60 % of the bandwidth of the 11 Mbps communication channel.			
Architectural decisions		**Sensitivity**	**Tradeoff**	**Risk**
AD 1 Choice for .NET remoting		S2		
AD 2 Agents located on machine controls AGV			T2	R2
AD 3 DynCNET protocol for transport assignment			T3	
AD 4 Two step deadlock prevention mechanism				R3
AD 5 Unicast communication in the middleware		S3		

Figure 5.32: Analysis architectural approaches with respect to bandwidth usage

remoting would no longer be usable. This potential problem was registered as sensitivity point S3 (see also S2).

5.6.4.3 Testing Communication Load

One important outcome of the ATAM evaluation was an improved insight on the tradeoff between flexibility and communication load. To further investigate this tradeoff, we conducted a number of tests after the ATAM workshop. Besides the simulation tests of the two approaches for transport assignment (see section 5.4), we tested the efficiency of the middleware in the AGV application by measuring bandwidth usage of a system in a real factory layout.

Fig. 5.33 shows the results of four consecutive test runs. We measured the

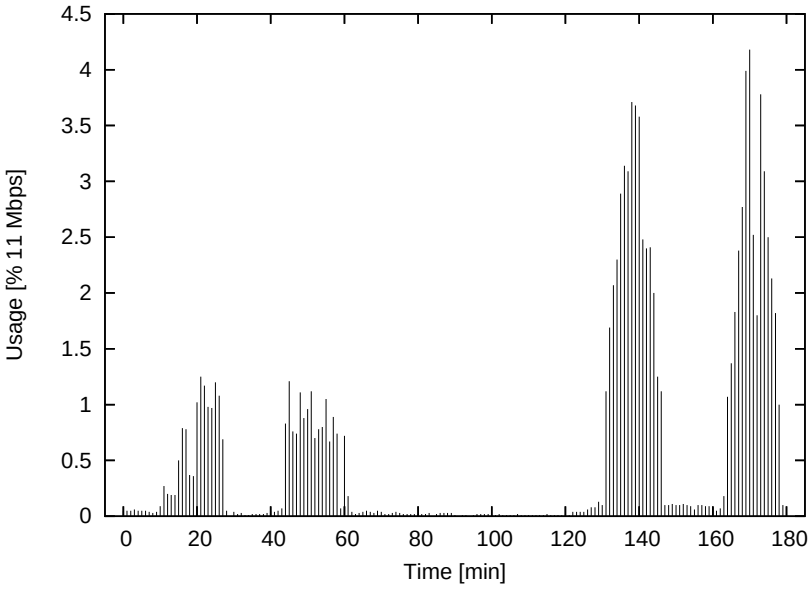


Figure 5.33: Bandwidth usage in a test setting

amount of data sent on the network by each AGV, and averaged this per minute to obtain the bandwidth usage relative to the bandwidth of a 11 Mbps IEEE 802.11 network. The first test (Time: 10–30 min.) has three AGVs, of which two were artificially put in deadlock (a situation which is avoided in normal operation), because then the collision avoidance protocol is continually restarted, and never succeeds. This is a peak load of the system. The second test (40–60 min.) has three AGVs driving around freely. The third test (130–150 min.) has five AGVs driving around freely. The fourth test (160–180 min.) has five AGVs, all artificially put in deadlock. During the time in between test runs, AGVs were repositioned manually. On average, the bandwidth usage doubles when going from three to five AGVs. This is because the AGVs need to interact relatively more to avoid collisions. Based on these test results, Egemin experts consider the bandwidth usage acceptable for an extrapolation to 12 AGVs, taking into account that the maximal bandwidth usage should be less than 60 % of the available 11 Mbps, and given that bandwidth optimizations were not applied yet.

5.6.5 Reflection on the ATAM Workshop

The ATAM workshop was a valuable experience. For the first time, the assembled group of stakeholders discussed the architecture in depth. Participants agreed

that their insights were improved on: (1) the importance of software architecture in software engineering; (2) the importance of business drivers for architectural design; (3) the importance of explicitly listing and prioritizing quality attributes with the stakeholders; (4) the strengths and weaknesses of the architecture and architectural approaches. One interesting outcome of the ATAM was the clarification of the tradeoff between flexibility and communication load. Although the architects were aware of this tradeoff, during the ATAM several architectural decisions were identified as risky and required further investigation. Field tests after the ATAM proved that the communication cost remains under control.

A number of critical reflections about the ATAM were made as well. (1) Performing a thorough and complete architectural evaluation using the ATAM is difficult to achieve in a single day. (2) Coming up with a quality attribute tree proved to be difficult, time consuming, and at times tedious. A lack of experience and clear guidelines of how to build up such a tree hindered and slowed down the discussion. (3) While the general AGV software architecture was developed with several automation projects in mind, the ATAM evaluated it within the scope of a single automation project. The ATAM however, is devised to evaluate a single architecture in a single project. This difference in scope hindered the discussions because some architectural decisions were motivated by the product line nature of the architecture. (4) During the preparation of the ATAM, we experienced a lack of good tool support to document architectures. Currently, drawing architectural diagrams and building up the architectural documentation incurs much overhead. Especially changing the documentation and keeping everything up-to-date (e.g. cross references and relations between different parts of the documentation) turned out to be hard and time consuming. Good tool support would be helpful.

5.6.6 Demonstrator of AGV Transportation System

As a proof of concept, we have developed a demonstrator of the decentralized AGV transportation system. The demonstrator with two AGVs is developed in .Net and supports the basic functionality for executing transport orders. The core of the action selection module of the AGVs is set up as a free-flow tree. A monitor enables remote access of the AGVs and generates a fusion view that represents the status of the local virtual environments of both AGVs. Fig. 5.34 shows a snapshot of the AGVs in action with the fusion view.

More information and demonstration movies of the prototype implementation can be found at the DistriNet website [6].

5.7 Concluding Remarks

The AGV transportation system provided a challenging application to investigate the feasibility of applying a situated multiagent system in an industrial setting.

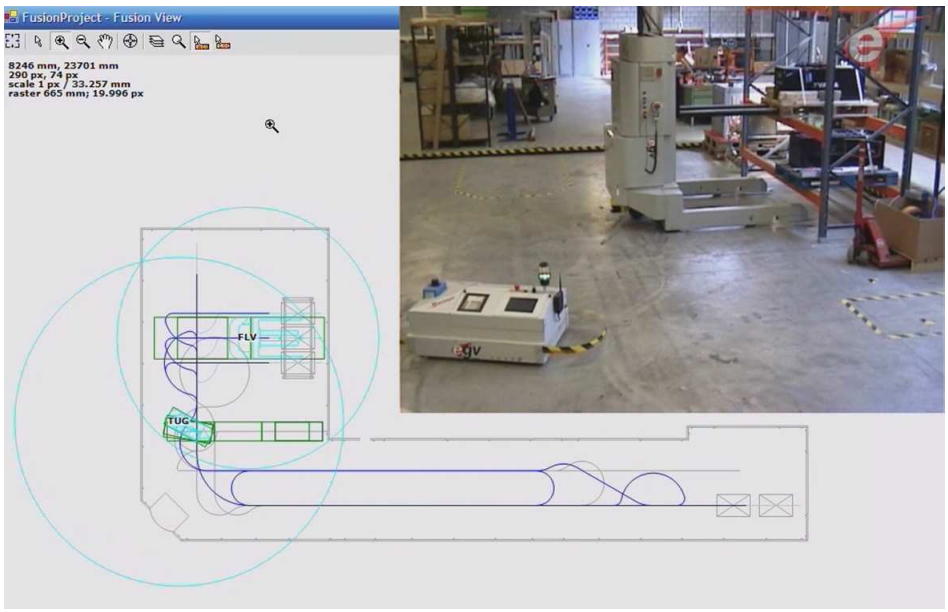


Figure 5.34: Demonstrator with AGVs in action

The architectural design of this complex real-world application and its successful implementation demonstrated the usefulness of situated multiagent systems to achieve important quality goals such as flexibility and openness.

Engineering the AGV application gained us a better insight in the important role of software architecture in the software engineering process. The AGV transportation system is a quite complex piece of software. Software architecture allowed us to concisely define how the software of the AGV application is structured and how its components work together to provide the required functionality and to achieve the main quality goals. The technical documentation of the software architecture specifies inviolable constraints as well as exploitable freedom to the developers of the software. We learned that software architecture can—and should!—be evaluated. Evaluating a software architecture brings the stakeholders together to discuss the software; and more importantly, it compels them to define and prioritize the quality requirements of the system precisely. Architecture evaluation makes explicit causal connections between design decisions made in the architecture and the qualities and properties in the software system. It reveals the weak and strong points of the architecture, providing valuable feedback to the architects. This helps to avoid problems later in the development process when changes in the structure of the software are much harder to achieve, or become too expensive.

Applying situated multiagent systems in this real-world application learned us a lot about the connection between multiagent systems and software architecture. The primary use of situated multiagent systems came from the way in which we *structured* the software as a set of self-managing agents that coordinate through an environment. In particular, the integrated set of mechanisms for adaptivity helped us to shape the software architecture of the AGV application and to provide the required functionalities and achieve the important quality goals flexibility and openness.

The insights derived from the architectural design of the AGV transportation system have considerably contributed to the development of the reference architecture for situated multiagent systems. We gained valuable insights about the structures and processes of the application environment, and the interaction of the situated multiagent system with the deployment context. These assets were used to structure the application environment of the reference architecture.

Finally, our project partner Egemin learned a lot about the potential as well as the possible implications of applying multiagent system technology in AGV systems. Moving from a centralized to a decentralized architecture implies a radical redesign which has a deep impact on the software and the organization. This calls for a step-wise integration of agent technology. At the time of writing this thesis, Egemin plans to apply the first results of the project in a system for one of its clients. In this application, Egemin experiences the need for improved flexibility and plans to integrate one of the transport assignment approaches in the system.

Chapter 6

Related Work

The two pillars in this work are software architecture and situated multiagent systems. In the text, we have extensively referred to work related to both these pillars. In chapter 2, we explained how our architecture-centric perspective on software engineering is related to other approaches in the domain. In chapter 3, we explained in detail how our model for situated multiagent systems relates to state-of-the-art work in the domain. We discussed related work on the concept of environment, and models and architectures for situated agents.

The core of this research however, is the integration of the two pillars in an architecture-centric perspective on software engineering with multiagent systems. Work related to this perspective is only mentioned briefly in the text. In this chapter, we discuss related work that explicitly considers the connection between software architecture and multiagent systems. The discussion is divided in two main topics. First, we discuss related work on architectural styles and multiagent systems. Then, we explain related work on reference models and architectures for multiagent systems. Additionally, we give a brief overview of related work on the control of AGV transportation systems. It is important to notice that the overview is not intended to be complete, our goal is to give a representative overview of related research.

6.1 Architectural Styles and Multiagent Systems

In this section, we discuss related work on quality attributes and architectural styles for multiagent systems.

Architectural Properties of Multiagent Systems. In [181], Shehory presents an initial study on the role of multiagent systems in software engineering, and in particular their merit as a software architecture style. The author observes that the largest part of research in the design of multiagent systems addresses

the question: given a computational problem, can one build a multiagent system to solve it? However, a more fundamental question is left unanswered: given a computational problem, is a multiagent system an appropriate solution? An answer to this question should precede the previous one, lest multiagent systems may be developed where much simpler, more efficient solutions apply.

The author presents an initial set of architectural properties that can support designers to assess the suitability of a multiagent system as a solution to a given problem. The properties provide a means to characterize multiagent systems as a software architecture style. Properties include the agent internal architecture, the multiagent system organization, the communication infrastructure and other infrastructure services such as a location service, security, and support for mobility. Starting from this perspective, the author evaluates a number of multiagent system frameworks and applications and compares them with respect to performance, flexibility, openness, and robustness.

Although the discussed properties are not unique to multiagent systems, the author states that the combination of the properties results in systems that are suitable for solving a particular family of problem domains. Characteristics of these domains are: distribution of information, location, and control; the environment is open and dynamically changing; and uncertainty is present. At the same time, the author points out that if only a few of these domain characteristics are present, it may be advisable to consider other architectures as solutions instead of a multiagent system.

Almost a decade later, the majority of researchers in agent-oriented software engineering still pass over the analysis whether a multiagent system is an appropriate solution for a given problem. Our research shares the position of Shehory. In particular: (1) a designer should consider a multiagent system as one of the possible architectural solutions to a problem at hand; and (2) the choice should be driven by the characteristics of the problem domain and the quality goals of the system.

Organizational Perspective on Multiagent Architectures. As part of the Tropos methodology [81], a set of architectural styles were proposed which adopt concepts from organization management theory [115, 58]. The styles are modelled using the $i^\star$ framework [239] which offers modelling concepts such as actor, goal, and actor dependency. Styles are evaluated with respect to various software quality attributes.

Proposed styles are joint venture, hierarchical contracting, bidding, and some others. As an example, the joint venture style models an agreement between a number of primary partner actors who benefit from sharing experience and knowledge. Each partner actor is autonomous and interacts directly with other partner actors to exchange services, data, and knowledge. However, the strategic operations of the joint venture are delegated to a joint management actor that coordinates tasks and manages the sharing of knowledge and resources. Secondary

partner actors supply services or support tasks for the organization core.

Different kinds of dependencies exist between actors, such as goal dependencies, task dependencies, and resource dependencies. An example of a task dependency in a joint venture is a coordination task between the joint management actor and a principal partner. A particular kind of dependency is the so-called softgoal that is used to specify quality attributes. Softgoal dependencies are similar to goal dependencies, but their fulfillment cannot be defined precisely [58]. [115] states that softgoals do not have a formal definition, and are amenable to a more qualitative kind of analysis. Examples of softgoals in the joint venture style are "knowledge sharing" among principle partner actors, and "added value" between the joint management actor and a principle partner actor. According to [58], a joint venture is particularly useful when adaptability, availability, and aggregability are important quality requirements. A joint venture is partially useful for systems that require predictability, security, cooperativity, and modularity. The style is less useful when competivity is an important quality goal.

A softgoal dependency in Tropos has no clear definition, it lacks a criterion to verify whether the goal is satisfied or not. Contrary, in our work we use quality attribute scenarios to precisely specify quality goals. A scenario includes a criterion to measure whether the scenario is satisfied. In [114], Klein et al. describe "general scenarios" that allow a precise articulation of quality attributes independent of a particular domain. This allows to specify scenarios for architectural styles. Another difference with our work is the handling of tradeoffs among quality requirements. Whereas we use a utility tree to prioritize quality requirements and to determine the drivers for architectural design, Tropos does not consider a systematic prioritization of quality goals such as a utility tree. In Tropos, a designer visualizes the design process and simultaneously attempts to satisfy the collection of softgoals for a system.

The assessment of architectural styles in Tropos is based on a set of quality attributes. Some of these attributes, such as availability and reliability, have been studied for years and have generally accepted definitions. Other attributes, such as cooperativity, competitivity, and aggregability, do not. Simply naming such attributes by themselves is not a sufficient basis on which to judge the suitability of an architectural style. Stating that the joint venture style shows aggregability is subject to interpretation and may lead to misunderstanding. In our work, we have rigorously specified the various architectural patterns of the reference architecture in different views, and explained the rationale for the design of each view.

Architectural Evaluation of Agent-Based Systems. In [234], Woods and Barbacci study the evaluation of quality attributes of architectures of agent-based systems in the context of ATAM. The authors put forward an initial list of four relevant quality attributes for agent-based systems. The first attribute is performance predictability. Due to the autonomy of agents, it is difficult to predict the overall behavior of the system. The second attribute is security. Verifying au-

thenticity for data access is an important concern of many agent-based systems. The third quality attribute is adaptability to changes in the environment. Agents are usually required to adapt to changes in their environment, including agents that leave the system and new agents that enter the system. The fourth attribute considered is availability. Availability of functionality is related to the presence of agents and other services in the system and their mutual dependencies.

To discuss quality attributes in agent-based systems, the authors propose a classification of agent-based systems. The classification abstracts from particular agent architectures, but focusses on the coordination among agents. The classification is inspired by previous work of Hayden et al. [89]. Example classes are matchmaker and broker that act as mediators between agents that provide services and agents that request for services. For each class, the authors define a set of quality attribute scenarios. Scenarios are formulated in a template form that consists of three parts: "may affect" describes the quality attributes that may be affected by the scenario; "implications" describes the risks or potential problems illuminated by the scenario; and "possible solutions" proposes ways to cope with possible risks. As an example, one of the scenarios of matchmaker is "provider fails after advertising service". This scenario may affect performance and reliability of the consumer of the service. A possible implication might be that the consumer blocks while it is waiting for the service, holding up the system. One possible solution is to let the consumer times out and notify the matchmaker.

The approach of Woods and Barbacci requires a decomposition of the agent based system into primitive fragments that fit the generic defined agent types (matchmaker, broker, etc.). Scenarios can then be specified based on the interaction between the identified fragments. However, the presented scenarios are generic and lack specificity. When applied to a real system such as the AGV application, scenarios should be further refined according to the domain specific requirements and constraints. In addition, the scenarios only support the evaluation of communicative interactions between the agents. For some domains this may cover a significant part of the system. However, for other domains such as the AGV application, direct communication takes up only a very small part of the system.

6.2 Reference Models and Architectures for Multiagent Systems

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

PROSA: Reference Architecture for Manufacturing Systems. [236] 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 [53, 193] 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. Contrary, the reference architecture for situated multiagent systems is formally specified. This avoids interpretation.

[96] presents an interesting extension of PROSA in which the environment is exploited to obtain BDI (Believe, Desire, Intention [162]) functionality for the various PROSA agents. To avoid the complexity of BDI-based models and the accompa-

nying 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 [52], and has recently further been elaborated by Parunak and Brueckner, see [152]. The use of the environment in the work of [96] 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.

Aspect-Oriented Agent Architecture. In [79], 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 [108]—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 [111]. 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 [57]) 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.

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 [110]. IBM has developed an architectural blueprint for autonomic computing [3]. 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.

A Reference Model for Multiagent Systems. In [139], 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 [93], Jade [38], and Retsina [189]. 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.

6.3 Scheduling and Routing of AGV Transportation Systems

The control of AGVs is subject of active research since the mid 1980s. Most of the research has been conducted in the domain of AI and robotics. Recently, a number of researchers have applied multiagent systems, yet, most of this work is applied in small-scale projects.

AI and Robotics Approaches. The problems of routing and scheduling of AGVs is different from conventional path finding and scheduling problems. Scheduling and routing of AGVs is a time-critical problem, while a graph problem usually is not. Besides, the physical dimensions of the AGVs and the layout of the map must be taken into account.

Roughly spoken, three kinds of methods are applied to solve the routing and scheduling problem. Static methods use a shortest path algorithm to calculate routes for AGVs, see e.g. [66]. In case there exists an overlap between paths of AGVs, only one AGV is allowed to proceed. The other AGVs have to wait until the first AGV has reached its destination. Such algorithms are simple, but not efficient. Time-window-based methods, maintain for each node in the layout a list of time-windows reserved by scheduled AGVs. An algorithm routes vehicles through the layout taking into account the reservation times of nodes, see e.g. [112]. Dynamic methods apply incremental routing. An example algorithm is given in [190]. This algorithm selects the next node for the AGV to visit (towards its destination) based on the status of the neighboring nodes (reserved or not) and the shortest travel time. This is repeated until the vehicle reaches its destination. Measurements show that the algorithm is significant faster than non-dynamic algorithms, yet, the calculated routes are less efficient.

A number of researchers have investigated learning techniques to improve scheduling and routing of AGVs, see e.g. [159, 124]. This latter work applies re-

inforcement learning techniques and demonstrates that the approach outperforms simple heuristics such as first-come-first-served and nearest-station-first.

Contrary to the decentralized approach we have applied in the EMC^2 project, traditional scheduling and routing algorithms usually run on a central traffic control system from where commands are dispatched to the vehicles [161]. Moreover, most approaches are intended to find an optimal schedule for a particular setting. Such approaches are very efficient when the tasks are known in advance as for example the loading and unloading of a ship in a container terminal. In our work, scheduling and routing are going concerns, with AGVs operating in a highly dynamic environment.

Multiagent System Approaches. [147] presents a decentralized approach for collision-free movements of vehicles. In this approach, agents use cognitive planning to steer the AGVs through the warehouse layout. [39] discusses a behavior-based approach for decentralized control of automatic guided vehicles. In this work, conflict resolution with respect to collision and deadlock avoidance is managed by the agents based on local information. In [121], Lindijer applies another agent-based approach to determine conflict-free routes for AGVs. The author motivates his approach by considering quality requirements, including safety, flexibility, and scalability. Central to the approach is the concept of semaphore that is used as a traffic control component that guards shared infrastructure resources in the system such as an intersection. The system is validated with simplified scale models of real AGVs.

Arora and his colleagues have published a number of papers that describe the control of AGV systems with an agent-based decentralized architecture [22, 23]. Vehicles select their own routes and resolve the conflicts that arise during their motion. Control laws are applied to find save conditions for AGVs to move.

[47] discusses a variation on the field-based approach where agents construct a field in their direct neighborhood to achieve routing and deadlock avoidance in a simplified AGV system. Hoshino et al. [97] study a transportation system in which cranes unload a container ship and pass the loads to AGVs that bring them to a storage depot. Each AGV and crane is represented in the system by an agent. The authors investigate various mechanisms for AGV agents to select a suitable crane agent. The selection mechanisms are based on the actual and local situation of AGVs and cranes, examples are selection based on distance, time, and area (quay, transportation, and storage). The selection mechanism are combined with random container storage and planned storage. Simulations allow to determine the optimal combination of cranes and AGVs for a particular throughput. The approach uses an off-line simulation to find an optimal solution in advance. Such approach is restricted to domains where no disturbances are expected.

Contrary to our research, the discussed agent-based approaches are only validated in simulations and under a number of simplifying assumptions. Applying decentralized control in a real industrial setting involves numerous complicating

factors that deeply affect the scheduling and routing of AGVs. Most of the related work focusses on isolated concerns in AGV control. For a practical application however—as the AGV application in the EMC^2 project, different concerns have to be integrated, which is not a trivial problem.

One lesson we learned from our experience is that communication is a major bottleneck in a decentralized AGV control system. Most related work only considers simple layouts with a small number of AGVs, and abstracts from communication costs.

An important difference between our research and the discussed approaches is that we have applied an architecture-centric design for the AGV application in the EMC^2 project. Scheduling and routing are integrated in the software architecture with other concerns such as deadlock avoidance and maintenance of the AGVs. Most related work does not consider software architecture explicitly. As a consequence, little attention is payed to the tradeoffs between quality goals. In the EMC^2 project on the other hand, the tradeoffs between quality goals were the drivers for the system design.

Chapter 7

Conclusions

The research presented in this dissertation started from the observation that developing and managing today's distributed applications is complex. We identified three important reasons for the increasing complexity that characterize the family of systems we target in our research: (1) stakeholders involved in the systems have various, often conflicting quality requirements; (2) the systems are subject to highly dynamic and changing operating conditions; (3) activity in the systems is inherently localized, global control is hard to achieve or even impossible.

In this dissertation, we presented an approach for developing such complex systems. The approach integrates situated multiagent systems in an architecture-centric software engineering process. Key aspects of the approach are architecture-centric software development, self-management, and decentralized control. These aspects enable to handle the complexity of the target family of applications as follows:

- *Quality goals and tradeoffs.* Putting software architecture at the heart of the software development process compels the architects and other stakeholders involved in the system to deal explicitly with quality goals and tradeoffs among the various requirements of the system.

- *Dynamic and changing operating conditions.* Self-management enables a software system to deal autonomously with the dynamic and changing circumstances in which it has to operate. Key qualities for endowing systems with abilities to manage dynamism and change are flexibility and openness. Situated agents adapt their behavior according to the changing situation in the environment, the multiagent system can cope with agents leaving the system and new agents that enter.

- *Inherent locality of activity.* Decentralized control is essential to cope with the inherent locality of activity. In a system where global control is not

an option, the functionality of the system has to be achieved by collaborating subsystems. Control in a situated multiagent system is decentralized, situated agents cooperate to achieve the overall functionality of the system.

In the remainder of this concluding section, we first summarize the main contributions of our research. Next, we point to a number of lessons we have learned from applying multiagent systems in an industrial setting. We conclude the thesis with suggestions for further research, and a closing reflection on software engineering with multiagent systems.

7.1 Contributions

In this dissertation, we presented a promising perspective on software engineering with multiagent systems. Whereas agent-oriented software engineering generally considers multiagent systems as a radically new way of engineering software, we presented an engineering approach which integrates multiagent systems as software architecture in a general software engineering process. We consider the structuring of a system as a set of agents embedded in an environment as an architectural approach that provides particular quality attributes. In our research, we have developed a reference architecture for situated multiagent systems. This reference architecture provides a reusable architectural approach to develop systems in which flexibility and openness are important quality goals.

The contribution of our research consists of three parts. First, we have developed an advanced model for situated multiagent systems that extends state-of-the-art approaches in the domain. Second, from our experiences with building various multiagent system applications, we have developed a reference architecture for situated multiagent systems. Third, we have validated the usefulness of architecture-centric software development with situated multiagent systems in practice. Concrete contributions of our research are:

- We have developed a new perspective on the role of the environment in multiagent systems [216, 223, 230, 208, 215]. In particular, we have promoted the environment to a first-class abstraction that can be exploited creatively in the design of multiagent system applications.

- We have extended state-of-the-art approaches in situated multiagent systems with an advanced model for situated agents [206, 228, 226, 184, 227] that supports selective perception, social behavior with roles and situated commitments, and protocol-based communication.

- From our experiences with building various situated multiagent system applications, we have developed a reference architecture for situated multiagent systems [214, 210, 213]. 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. To demonstrate the feasibility of the reference architecture, we have developed an object-oriented framework that implements the architecture, and we have instantiated the framework for a couple of prototype applications.

- We have applied a situated multiagent system in a challenging industrial automated transportation system [222, 221, 217, 177]. The insights derived from the architectural design of this application has considerably contributed to the development of the reference architecture. The architectural design, the development, and the evaluation of this complex application proved the feasibility of situated multiagent systems in a real-world setting [198, 211, 45].

7.2 Lessons Learned from Applying Multiagent Systems in Practice

Although multiagent systems have been studied for more than two decades, industrial applications remain rare [156, 194]. The lack of experiences with applying agent technology in practice hampers the maturation of the domain. In this section, we report some lessons we learned from applying multiagent systems in a complex real-world application.

Qualities and tradeoffs. A main motivation for applying a multiagent system to the AGV transportation system was to investigate whether the decentralized architecture could improve flexibility and openness. Obviously, a commercial product such as an AGV transportation system is subject to various quality requirements and business constraints. The decentralized architecture introduces new tradeoffs between the various system requirements.

Important lessons we learned are: (1) the motivation to apply a multiagent system should be driven by quality goals; (2) we experienced a tendency from our industrial partner to overestimate agent technology (e.g., adding new capabilities to an autonomous agent—such as integrating functionality to manoeuvre an AGV around an obstacle—was simplified). It is therefore important to deal explicitly with tradeoffs between system requirements from the early start of a project.

Integration with legacy systems. Most industrial software systems require an integration with legacy systems, and this was the case for the AGV transportation system as well. Egemin has developed a common framework that provides basic support for various concerns such as logging, persistency, and security. This framework is used over the various software systems that are developed by the company. Obviously, the AGV application software has to be integrated with the framework. Besides, various parts of the existing AGV control software could be

reused for the decentralized architecture, examples are the low-level control software for AGVs, the layout of maps, and the basic routing algorithm. Reusing this software saved a lot of work.

Lessons we learned are: (1) the integration with legacy software is a matter of fact when agent technology is applied in an industrial setting; (2) software architecture provides the means to reason about, and deal with the integration of legacy software in an agent-based system.

Stepwise integration. The existing AGV transportation systems deployed by Egemin have a centralized architecture. The agent-based architecture on the other hand has a decentralized architecture. Switching towards an agent-based architecture is a big step with far reaching effects for the company, not only for the software but for the whole organization.

A lesson we learned is: integration of an agent-based approach should be done in a controlled way, step-by-step. For example, in a coming project, Egemin experiences the need for improved flexibility and plans to integrate the agent-based approach for transport assignment in their usual architecture.

Evaluation. The ATAM evaluation of the software architecture contributed to a better understanding of the strengths and weaknesses of the decentralized architecture. Besides the evaluation of quality attributes, the functional behavior of the system must be evaluated. It is well known that giving guarantees about the global behavior of a decentralized system is hard. Distributed algorithms are complex, hard to debug, and difficult to evaluate (especially in a physical setting).

Lessons we learned are: (1) a disciplined evaluation of the software architecture of the agent-based system is invaluable; (2) debugging a decentralized system is hard; (3) thorough simulations are the main vehicle to give (to a certain extent) guarantees about global properties of the system.

7.3 Future work

We give a number of suggestions for future research.

Architectural design with a reference architecture. Research and current practice in software architecture mainly focus on the documentation, design, and evaluation of software architectures for concrete applications. A software product line is a relatively new idea that aims to collect a common set of core assets that can be used to develop a specific family of software products [62]. A product line architecture specifies a common core that is used for all products of the family and variation points that provide variability to tailor each product to its specific needs. Various methods have been studied and developed for documenting, designing, and evaluating product line architectures, see e.g., [134, 145].

A reference architecture, on the other hand, describes a blueprint architecture that can be used to develop concrete software architectures in a particular domain. As such, a reference architecture can not be documented, applied, and evaluated in the same way as a concrete software architecture or a product line architecture.

Researchers and engineers use various methods to document, apply, and evaluate reference architectures. Two illustrative examples are [36] and [26]. [36] discusses the documentation of a reference architecture with views. The author emphasizes the identification of interfaces of architectural elements and the specification of adaptation guidelines for instructing architects how to instantiate target architectures from the reference architecture. [26] proposes an integrated approach for documenting, applying, and evaluating a reference architecture that is based on a combination of the IEEE 1471 standard for "Recommended Practice for Architectural Description of Software-Intensive Systems"[99], the Rational Unified Process [118], and the Unified Modelling Language [13].

In our research, we have documented the reference architecture for situated multiagent systems with a set of architectural views that specify the architectural core and variation mechanisms. Our approach joins the suggestion of UP [118, 100] to document a reference architecture with different views. UP however, is unclear on how to specify the variability of a reference architecture, which we have documented with variation mechanisms. To design a concrete software architecture, we apply a process of iterative decomposition inspired by attribute driven design (ADD). The reference architecture can serve as a blueprint to *guide* the architect through the decomposition process. It provides an integrated set of architectural patterns the architect can draw from to select suitable architectural solutions. To refine and extend specific architectural elements, additional common architectural patterns have to be selected by the architect.

It would be interesting to compare our approach to document and apply a reference architecture with other approaches. The insights derived from such a study can be used to develop common methods to document a reference architecture, to apply a reference architecture in architectural design, and to evaluate a reference architecture.

Software architecture and multiagent systems. The connection between software architecture and multiagent systems provides a promising venue for future research.

In our research, we have put forward flexibility and openness as important qualities to apply situated multiagent systems. However, multiagent systems are generally considered to be useful for other qualities as well, such as robustness and scalability. It would be interesting to investigate how these qualities translate to architectural approaches and how these qualities tradeoff with other qualities in the system.

The reference architecture for situated multiagent systems abstracts from the concrete deployment of the multiagent system application. For a distributed appli-

cation, the deployment context consists of multiple processors deployed on different nodes that are connected through a network. For such applications, agents and the application environment typically have to be distributed over the processors of the application nodes. It would be interesting to investigate whether patterns can be identified for particular classes of applications that allow to specify distribution as a concern of multiagent systems.

The modularization of the reference architecture is primarily based on the separation of different functionalities of agents (perception, communication, actions) and their support in the environment. An important advantage of this modularization is that the structure of the architecture is easy to understand. Yet, the integration of concerns such as security and logging that crosscut different modules of the architecture does not naturally fit with the basic modularization. Aspect-oriented software development (AOSD) provides an interesting approach to modularize and integrate crosscutting concerns in a system. It would be interesting to investigate how AOSD can be used to integrate crosscutting concerns in the reference architecture.

Variation mechanisms in the reference architecture only provide high-level guidance of how particular parts of a software architecture can be tailored to the specific needs of the application at hand. Additional support for the design of particular mechanisms would be helpful for architects and developers. Examples are support for protocol-based communication, support for the design of behavior-based action selection mechanisms with roles and situated commitments (as for example in [184]), support for indirect interaction mechanisms, and the concrete design of laws.

Another interesting challenge is to develop a scientific foundation and techniques for verifying global behavior of decentralized multiagent systems. Such verification should provide sufficient information of global properties during architectural design. Interesting work in this direction is [238, 232].

7.4 Closing Reflection

As a specific domain of software systems ages and matures, more and more systems are built, and their functionality, structure, and behavior become common knowledge. At a software-architecture level, this kind of common knowledge is called a reference architecture.

Within the domain of situated multiagent systems, software systems have been studied and built for over two decades. The efforts of many prominent researchers have laid the foundation on which our research contributions are built. As such, the reference architecture for situated multiagent systems reifies the knowledge and expertise we have acquired during our research that is founded on two decades of domain maturing.

The reference architecture for situated multiagent systems demonstrates how

knowledge and practices with multiagent systems can systematically be documented and maturated in a form that has proven its value in mainstream software engineering. We believe that the integration of multiagent systems as software architecture in mainstream software engineering is a key to industrial adoption of multiagent systems.

Bibliography

[1] Automatic Guided Vehicle Simulator, DistriNet, K.U.Leuven, (6/2006). www.cs.kuleuven.ac.be/∼distrinet/taskforces/agentwise/agvsimulator/.

[2] Autonomic Computing: IBM's Perspective on the State of Information Technology, (6/2006). www.research.ibm.com/autonomic/research/.

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

[4] Microsoft Dynamic Systems Initiative Overview White Paper, (6/2006). www.microsoft.com/windowsserversystem/dsi/dsiwp.mspx.

[5] Architecture Description Languages, Software Engineerin Institute, CMU, (8/2006). www.sei.cmu.edu/str/descriptions/adl_body.html.

[6] DistriNet Research Group, Egemin Modular Controls Concept Project, (8/2006). www.cs.kuleuven.ac.be/cwis/research/distrinet/public/research/.

[7] 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/.

[8] FOLDOC: Free On-Line Dictionary of Computing, Imperial College London, (8/2006). http://foldoc.doc.ic.ac.uk/foldoc/index.html.

[9] H. V. D. Parunak, Chief Scientist NewVectors LLC, USA, Home Page, (8/2006). http://www.newvectors.net/staff/parunakv/.

[10] Lego Mindstorms, (8/2006). http://mindstorms.lego.com/.

[11] LeJOS, Lego Java Operating System for the Lego Mindstorms RCX, (8/2006). http://lejos.sourgeforce.com/.

[12] Software Engineering Institute, Carnegie Mellon University, (8/2006). http://www.sei.cmu.edu/.

[13] The Unified Modeling Language, (8/2006). http://www.uml.org/.

[14] P. Agre and D. Chapman. Pengi: An Implementation of a Theory of Activity. In *National Conference on Artificial Intelligence, Seattle, WA*, 1987.

[15] P. Agre and D. Chapman. What are Plans for? *Designing Autonomous Agents, MIT Press*, 1990.

[16] T. Al-Naeem, I. Gorton, M. Babar, F. Rabhi, and B. Benatallah. A Quality-driven Systematic Approach for Architecting Distributed Software Applications. In *27th International Conference on Software Engineering*, New York, NY, USA, 2005. ACM Press.

[17] J. Allen and G. Ferguson. Actions and Events in Interval Temporal Logic. *Journal of Logic and Computation, Special Issue on Actions and Processes*, 4:531–579, 1994.

[18] M. Arbib. Schema Theory. *Encyclopedia of Artificial Intelligence*, 1992.

[19] R. Arkin. Motor Schema-Based Mobile Robot Navigation. *International Journal of Robotics Research*, 8(4):92–112, 1989.

[20] R. Arkin. Integrating Behavioral, Perceptual, and World Knowledge in Reactive Navigation. *Designing Autonomous Agents, MIT Press*, 1990.

[21] R. Arkin. *Behavior-Based Robotics*. Massachusetts Institute of Technology, MIT Press, Cambridge, MA, USA, 1998.

[22] S Arora, A. Raina, and A. Mittal. Collision Avoidance Among AGVs at Junctions. In *IEEE Intelligent Vehicles Symposium*, 2000.

[23] S Arora, A. Raina, and A. Mittal. Hybrid Control in Automated Guided Vehicle Systems. In *IEEE Conference on Intelligent Transportation Systems*, 2001.

[24] C. Atkinson and T. Kuhne. Aspect-Oriented Development with Stratified Frameworks. *IEEE Software*, 20(1):81–89, 2003.

[25] J. Austin. *How To Do Things With Words*. Oxford University Press, Oxford, UK, 1962.

[26] P. Avgeriou. Describing, Instantiating and Evaluating a Reference Architecture: A Case Study. *Enterprise Architect Journal*, 2003. Fawcette Technical Publications.

[27] O. Babaoglu, H. Meling, and A. Montresor. Anthill: A Framework for the Development of Agent-Based Peer-to-Peer systems. In *22nd International Conference on Distributed Computing Systems*, Vienna, Austria, 2002. IEEE Computer Society, Digital Library.

[28] F. Bachmann and L. Bass. Managing Variability in Software Architectures. In *Symposium on Software Reusability*, New York, NY, USA, 2001. ACM Press.

[29] S. Bandini, M. L. Federici, S. Manzoni, and G. Vizarri. Towards a Methodology for Situated Cellular Agent Based Crowd Simulations. In *6th International Workshop on Engineering Societies in the Agents World, ESAW*, 2005.

[30] S. Bandini, S. Manzoni, and C. Simone. Dealing with Space in Multiagent Systems: A Model for Situated Multiagent Systems. In *1st International Joint Conference on Autonomous Agents and Multiagent Systems*. ACM Press, 2002.

[31] S. Bandini, S. Manzoni, and G. Vizzari. MultiAgent Approach to Localization Problems: the Case of Multilayered Multi Agent Situated System. *Web Intelligence and Agent Systems*, 2(3):155–166, 2004.

[32] S. Bandini, S. Manzoni, and G. Vizzari. A Spatially Dependent Communication Model. In *1st International Workschop on Environments for Multiagent Systems*, Lecture Notes in Computer Science, Vol. 3374. Springer-Verlag, 2005.

[33] M. Barbacci, R. Ellison, A. Lattanze, J. Stafford, C. Weinstock, and W. Wood. uality Attribute Workshops. Technical Report CMU/SEI-2003-TR-016, Software Engineering Institute, Carnegie Mellon University, PA, USA, 2003.

[34] M. Barbacci, M. Klein, T. Longstaff, and C. Weinstock. uality Attribute Workshops. Technical Report CMU/SEI-95-TR-21, Software Engineering Institute, Carnegie Mellon University, PA, USA, 1995.

[35] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley Publishing Comp., 2003.

[36] J. Batman. Characteristics of an Organization with Mature Architecture Practices. *Essays on Software Architecture*, (6/2006). Software Engineering Institute, http://www.sei.cmu.edu/architecture/essays.html.

[37] K. Beck and R. Johnson. Patterns Generate Architectures. In *ECOOP '94: Proceedings of the 8th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Vol. 821, London, UK, 1994. Springer-Verlag.

[38] 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.

[39] S. Berman, Y. Edan, and M. Jamshidi. Decentralized autonomous AGVs in material handling. *Transactions on Robotics and Automation*, 19(4), 2003.

[40] C. Bernon, M-P. Gleizes, S. Peyruqueou, and G. Picard. Adelfe: A Methodology for Adaptive Multiagent Systems Engineering. In *3th International Workshop on Societies in the Agents World, ESAW*, Lecture Notes in Computer Science, Vol. 2577. Springer-Verlag, 2002.

[41] E. Bonabeau, F. Henaux, S. Guérin, D. Snyers, P. Kuntz, and G. Theraulaz. Routing in telecommunications networks with ant-like agents. In *Second International Workshop on Intelligent Agents for Telecommunication a Applications, IATA*, Paris, France, 1998. Springer-Verlag.

[42] N. Boucké. *Situated Multiagent System Approach for Distributing Control in Automatic Guided Vehicle Systems.* Master Thesis, Katholieke Universiteit Leuven, Belgium, 2003.

[43] N. Boucké, T. Holvoet, T. Lefever, R. Sempels, K. Schelfthout, D. Weyns, and J. Wielemans. Applying the Architecture Tradeoff Analysis Method to an Industrial Multiagent System Application. In *Technical Report CW 431.* Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 2005.

[44] 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.

[45] 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.

[46] C. Boutilier and R. I. Brafman. Partial-order planning with concurrent interacting actions. *Journal on Artificial Intelligence Research*, 14:105–136, 2001.

[47] L. Breton, S. Maza, and P. Castagna. Simulation multi-agent de systèmes d'AGVs: comparaison avec une approche prédictive. 5^e *Conférence Francophone de Modélisation et Simulation*, 2004.

[48] M. Brodie, I. Rish, S. Ma, and N. Odintsova. Active probing strategies for problem determination. In *18th International Joint Conference on Artificial Intelligence*, 2003.

[49] R. Brooks. Achieving artificial intelligence through building robots. *AI Memo 899, MIT Lab*, 1986.

[50] R. Brooks. The Behavior Language; User's Guide. *AI Memo 1227, MIT Lab*, 1990.

[51] R. Brooks. Intelligence without reason. In *12th International Joint Conference on Artificial Intelligence*, Sydney, Australia, 1991.

[52] S. Brueckner. *Return from the Ant, Synthetic Ecosystems for Manufacturing Control*. Ph.D Dissertation, Humboldt University, Berlin, Germany, 2000.

[53] 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.

[54] J. Bryson. *Intelligence by Design, Principles of Modularity and Coordination for Engineering Complex Adaptive Agents*. PhD Dissertation, MIT, USA, 2001.

[55] F. Buchmann and L. Bass. Introduction to the Attribute Driven Design Method. In *23rd International Conference on Software Engineering*, Toronto, Ontario, Canada, 2001. IEEE Computer Society.

[56] G. Cabri, L. Ferrari, and F. Zambonelli. Role-Based Approaches for Engineering Interactions in Large-Scale Multi-agent Systems. In *Software Engineering for Multi-Agent Systems II*, Lecture Notes in Computer Science, Vol. 2940. Springer-Verlag, 2004.

[57] 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.

[58] J. Castro, M. Kolp, and J. Mylopoulos. Towards Requirements-Driven Information Systems Engineering: The Tropos Project. *Informatica Systems*, 27(6):365–389, 2002.

[59] L. Claesen. *Regional Synchronization in Situated Multiagent Systems*. Master Thesis, Katholieke Universiteit Leuven, Belgium, 2004.

[60] 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.

[61] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison Wesley Publishing Comp., 2002.

[62] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Publishing Comp., August 2001.

[63] P. Cohen and H. Levesque. Teamwork. *Nous, Special Issue on Cognitive Science and Artificial Intelligence*, 4(25):487–512, 2002.

[64] C. Cuesta, M. del Pilar Romay, P. de la Fuente, and M. Barrio-Solorźano. Architectural Aspects of Architectural Aspects. In *2nd European Wirkshop on Software Architecture, EWSA*, Lecture Notes in Computer Science, Vol. 3527. Springer, 2005.

[65] R. Custers. *The Agent Network Architecture Extended for Cooperating Robots*. Master Thesis, Katholieke Universiteit Leuven, Belgium, 2003.

[66] D. Daniels. Real-time Conflict Resolution in Automated Guided Vehicle Scheduling. *PhD Dissertation: Dept. of Industrial Eng., Penn. State University, USA*, 1988.

[67] B. Demarsin. *DynCNET: A Protocol for Flexible Transport Assignment in AGV Transportation Systems*. Master Thesis, Katholieke Universiteit Leuven, Belgium, 2006.

[68] Y. Demazeau. Multi-Agent Systems Methodology. In *2nd Franco-Mexican School on Cooperative and Distributed Systems, LAFMI 2003, http://lafmi.lania.mx/escuelas/esd03/ponencias/Demazeau.pdf*.

[69] J. Deneubourg and S. Goss. Collective Patterns and Decision Making. *Ecology, Ethology and Evolution*, 1:295–311, 1989.

[70] M. Dorigo and L. Gambardella. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[71] B. Dunin-Keplicz and R. Verbrugge. Calibrating collective commitments. In *Multi-Agent Systems and Applications III, 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS, Prague, Czech Republic*, Lecture Notes in Computer Science, Vol. 2691. Springer, 2003.

[72] E. Durfee and V. Lesser. Negotiating Task Decomposition and Allocation Using Partial Global Planning. *Distributed Artificial Intelligence*, 2:229–244, 1989.

[73] M. Fayad and D. Schmidt. Object-Oriented Application Frameworks, Guest Editorial. *Communications of the ACM, Special Issue on Object-Oriented Application Frameworks*, 40(10):32–38, 1997.

[74] J. Ferber. *An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, 1999.

[75] J. Ferber, F. Michel, and J. Baez. AGRE: Integrating environments with organizations. In *1st International Workschop on Environments for Multiagent Systems*, Lecture Notes in Computer Science, Vol. 3374. Springer-Verlag, 2005.

[76] 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.

[77] FIPA. Foundation for Intelligent Physical Agents, FIPA Abstract Architecture Specification. *http://www.fipa.org/repository/bysubject.html*, (8/2006).

[78] B. Gallagher. Using the Architecture Tradeoff Analysis Method to Evaluate a Reference Architecture. Technical Report CMU/SEI-2000-TN-007, Software Engineering Institute, Carnegie Mellon University, PA, USA, 2000.

[79] 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.

[80] M. Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmanns, 1997.

[81] F. Giunchiglia, J. Mylopoulos, and A. Perini. *The TROPOS Software Development Methodology: Processes, Models and Diagrams.* 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems AAMAS'02, ACM Press, New York, 2002.

[82] O. Glorieux. *A Model for Adaptive Agents, Applied to the Packet–World.* Master Thesis, Katholieke Universiteit Leuven, Belgium, 2003.

[83] P. Grassé. La Reconstruction du nid et les Coordinations Inter-Individuelles chez Bellicositermes Natalensis et Cubitermes sp. La theorie de la Stigmergie. Essai d'interpretation du Comportement des Termites Constructeurs. *Insectes Sociaux*, 6:41–81, 1959.

[84] N. Griffiths, M. Luck, and M. d'Iverno. Cooperative Plan Annotation through Trust. In *UK Workshop on Multi-Agent Systems*, Liverpool, UK, 2002.

[85] M. Griss, I. Jacobson, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success.* Addison Wesley Professional, 1997.

[86] S. Hadim and N. Mohamed. Middleware Challenges and Approaches for Wireless Sensor Networks. *IEEE Distributed Systems Online*, 7(3), 2006.

[87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 3(8):231–274, 1987.

[88] P. Hart, N. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):28–29, 1968.

[89] S. Hayden, C. Carrick, and Q. Yang. A Catalog of Agent Coordination Patterns. In *3th International Conference on Autonomous Agents*, New York, NY, USA, 1999. ACM Press.

[90] A. Helleboogh, T. Holvoet, D. Weyns, and Y. Berbers. Extending time management support for multi-agent systems. In *Multiagent and Multiagent-based Simulation, New York, USA*, Lecture Notes in Computer Science, Vol. 3415, 2005.

[91] A. Helleboogh, T. Holvoet, D. Weyns, and Y. Berbers. Towards time management adaptability in multi-agent systems. In *Agents and Multiagent Systems III: Adaptation and Multiagent Learning*, Lecture Notes in Computer Science, Vol. 3494, 2005.

[92] E. Helsen and K. Deschacht. *The Delta Framework for Situated Multiagent Systems*. Master Thesis, Katholieke Universiteit Leuven, Belgium, 2005.

[93] 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.

[94] A. Holvoet. Visualisation of a Peer-to-Peer Network. *Master Thesis, Katholieke Universiteit Leuven, Belgium*, 2004.

[95] T. Holvoet and E. Steegmans. Application-Specific Reuse of Agent Roles. In *Software Engineering for Large-Scale Multi-Agent Systems*, Lecture Notes in Computer Science, Vol. 2603. Springer-Verlag, New York, 2003.

[96] 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.

[97] S. Hoshino, J. Ota, A. Shinozaki, and H. Hashimoto. Design of an AGV Transportation System by Considering Management Model in an ACT. *Intelligent Autonomous Systems*, 9:505–514, 2006.

[98] M. Huhns and L.M. Stephens. *Multiagent Systems and Societies of Agents. In Multiagent Systems, A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 2000.

[99] IEEE. Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE Std 1471-2000. *Institute of Electrical and Electronics Engineers*, 2000.

[100] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[101] M. Jazayeri, A. Ran, and F. van der Linden. *Software Architecture for Product Families*. Addison Wesley Longman Inc., 2000.

[102] N. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, 2001.

[103] N. Jennings, A. Luomuscio, S. Parsons, C. Sierra, and M. Wooldridge. Automated Negotiation: Prospects, Methods and Challenges. *International Journal of Group Decision and Negotiation*, 10(2):199–215, 2001.

[104] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object Oriented Programming*, 1(2):22–35, 1988.

[105] L. Kaelbling. Goals as Parallel Program Specifications. In *7th National Conference on Artifical Intelligence, Minneapolis, Minnesota*, 1988.

[106] L. Kaelbling and J. Rosenschein. Action and Planning in Embedded Agents. *Designing Autonomous Agents, MIT Press*, 1990.

[107] E. Kendall. Role modeling for agent system analysis, design, and implementation. *IEEE Concurrency*, 8(2):34–41, 2000.

[108] E. Kendall and C. Jiang. Multiagent System Design Based on Object Oriented Patterns. *Journal of Object Oriented Programming*, 10(3):41–47, 1997.

[109] J. Kephart. Research Challenges of Autonomic Computing (IBM). *Invited talk, International Conference on Software Engineeging, St. Louis, USA*, 2005.

[110] J. Kephart and D. Chess. The Vision of Autonomic Computing. *IEEE Computer Magazine*, 36(1).

[111] 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.

[112] C. Kim and J. Tanchoco. Operational Control of a Bi-directional Automated Guided Vehicle Systems. *International Journal of Production Research*, 31(9):2123–2138, 2002.

[113] D. Kinny, M. Ljundberg, and A. Rao. Planning with Team Activity. In *4th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Lecture Notes in Computer Science, Vol. 830. Springer-Verlag, London, UK, 1992.

[114] M. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, and H. Lipson. Attribute-Based Architecture Styles. In *1st Working Conference on Software Architecture, WICSA*, San Antonio, TX, USA, 1999.

[115] M. Kolp, P. Giorgini, and J. Mylopoulos. A Goal-Based Organizational Perspective on Multi-agent Architectures. In *8th International Workshop on Intelligent Agents*, London, UK, 2002. Springer-Verlag.

[116] D. Kotz and R. Gray. Mobile Agents and the Future of the Internet. *ACM Operating Systems Review*, 33(3):3–17, 1999.

[117] P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, 1995.

[118] P. Kruchten. *The Rational Unified Process.* Addison Wesley Publishing Comp., 2003.

[119] Y. Labrou. Standardizing Agent Communication. New York, NY, USA, 2001. Springer-Verlag.

[120] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design.* Prentice Hall, 2002.

[121] D. Lindeijer. Controlling Automated Traffic Agents. *PhD Dissertation: University of Delft, The Netherlands*, 2003.

[122] P. Maes. Situated Agents can have Goals. *Designing Autonomous Agents, MIT Press*, 1990.

[123] P. Maes. Modeling Adaptive Autonomous Agents. *Artificial Life Journal*, 1(1-2):135–162, 1994.

[124] R. Makar, S. Mahadevan, and M. Ghavamzadeh. Hierarchical MultiAgent Reinforcement Learning. In *5th International Conference on Autonomous Agents*, 2001.

[125] C. Malcolm and T. Smithers. Symbol Grounding via a Hybrid Architecture in an Autonomous Assembly System. *Designing Autonomous Agents, MIT Press*, 1990.

[126] M. Mamei and F. Zambonelli. Co-Fields: A Physically Inspired Approach to Distributed Motion Coordination. *IEEE Pervasive Computing*, 3(2):52–61, 2004.

[127] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *2nd International Conference on Pervasive Computing and Communications*. IEEE Computer Society, Washington, DC, USA, 2004.

[128] M. Mamei and F. Zambonelli. *Field-based Coordination for Pervasive Multiagent Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[129] M. Mamei and F. Zambonelli. Motion Coordination in the Quake3 Arena Environment. In *Environments for Multiagent Systems, E4MAS*, Lecture Notes in Computer Science, Vol. 3374. Springer, 2005.

[130] M. Mamei, F. Zambonelli, and L. Leonardi. Distributed Motion Coordination with Co-Fields: A Case Study in Urban Traffic Management. In *6th IEEE Symposium on Autonomous Decentralized Systems, Pisa, Italy*. IEEE Press, 2003.

[131] M. Mamei, F. Zambonelli, and L. Leonardi. Tuples On The Air: A Middleware for Context-Aware Computing in Dynamic Networks. *International Conference on Distributed Computing Systems Workshops*, 2003.

[132] X. Mao and E. Yu. Organizational and social concepts in agent oriented software engineering. In *Agent-Oriented Software Engineering V, 5th International Workshop, AOSE, New York, NY, USA*, Lecture Notes in Computer Science, Vol. 3382. Springer-Verlag, 2004.

[133] M. Mataric. Leaning to Behave Socially. In *From Animals to Animats, 3th International Conference on Simulation of Adaptive Behavior*. MIT Press, 1994.

[134] M. Matinlassi. Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA. In *ICSE '04: 26th International Conference on Software Engineering*, Edinburgh, UK, 2004. IEEE Computer Society.

[135] S. McConell. Rapid Development: Taming Wild Software Schedules. *Microsoft Press*, 1996.

[136] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[137] J. De Meulenaere. *Stigmergy Applied in the Packet–World*. Master Thesis, Katholieke Universiteit Leuven, Belgium, 2004.

[138] F. Michel, A. Gouaich, and J. Ferber. Weak Interaction and Strong Interaction in Agent Based Simulations. In *Multi-Agent-Based Simulation III 4th International Workshop, MABS, Melbourne, Australia*, Lecture Notes in Computer Science, Vol. 2927. Springer-Verlag, 2003.

[139] 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.

[140] A. Murphy, G.P. Picco, and G.C. Roman. LIME: a Middleware for Physical and Logical Mobility. *21th International Conference on Distributed Computing Systems*, 2001.

[141] N. Nilsson. Teleo-Reactive Programs for Agent Control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.

[142] L. Northrop. A Framework for Software Product Line Practice, Version 4.2. *http://www.sei.cmu.edu/productlines/framework.html*, (8/2006).

[143] J. Odell, H. V. D. Parunak, and M. Fleischer. The Role of Roles. *Journal of Object Technology*, 2(1):39–51, 2003.

[144] J. Odell, H. V. D. Parunak, M. Fleischer, and S. Breuckner. Modeling Agents and their Environment. In *Agent-Oriented Software Engineering III, Third International Workshop, Bologna, Italy, 2002*, Lecture Notes in Computer Science, Vol. 2935. Springer-Verlag, 2003.

[145] F. Olumofin and V. Misic. Extending the ATAM Architecture Evaluation to Product Line Architectures. In *5th IEEE-IFIP Conference on Software Architecture*, Pittsburgh, Pennsylvania, USA, 2005.

[146] L. Padgham and M. Winikoff. Prometheus: A Methodology for Developing Intelligent Agents. In *Agent-Oriented Software-Engineering III*, Lecture Notes in Computer Science, Vol. 2585. Springer-Verlag, New York, 2003.

[147] L. Pallottino, V. G. Scordio, E. Frazzoli, and A. Bicchi. Decentralized cooperative conflict resolution for multiple nonholonomic vehicles. In *AIAA Conference on Guidance, Navigation and Control*, 2005.

[148] F. De Paoli and G. Vizzari. Context dependent management of field diffusion: an experimental framework. *Workshop Dagli Oggetti agli Agenti, Villasimius, Italy*, 2002.

[149] D. Parnas. On a "Buzzword": Hierarchical Structure. *Software pioneers: Contributions to software engineering*, pages 429–440, 2002.

[150] H. V. D. Parunak. Go to the Ant: Engineering Principles from Natural Agent Systems. *Annals of Operations Research*, 75:69–101, 1997.

[151] H. V. D. Parunak and S. Brueckner. Analyzing Stigmergic Learning for Self-Organizing Mobile Ad-Hoc Networks (MANET's). In *Engineering Self-Organising Systems, Methodologies and Applications, ESOA*, Lecture Notes in Computer Science, Vol. 3464. Springer, 2005.

[152] 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.

[153] H. V. D. Parunak, S. Brueckner, M. Fleischer, and J. Odell. A preliminary taxonomy of multiagent interactions. In *Agent-Oriented Software Engineering IV, 4th International Workshop, AOSE, Melbourne, Australia, 2003*, Lecture Notes in Computer Science, Vol. 2935. Springer-Verlag, 2004.

[154] H. V. D. Parunak, S. Brueckner, and J. Sauter. Digital Pheromones for Coordination of Unmanned Vehicles. In *Environments for Multiagent Systems, E4MAS*, Lecture Notes in Computer Science, Vol. 3374. Springer, 2005.

[155] H. V. D. Parunak, S. Brueckner, J. Sauter, and R. Matthews. Global Convergence of Local Agent Behaviors. In *4th Joint Conference on Autonomous Agents and Multiagent Systems*, Utrecht, The Netherlands, 2005.

[156] M. Pechouchek, D. Steiner, and S. Thompson. *Proceedings of the Industry Track of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems*. ACM, Utrecht, The Netherlands, 2005.

[157] D. Perry and A. Wolf. Foundations for the Study of Software Architecture. *Software Engineering Notes*, 17(2):40–52, 2000.

[158] E. Platon, N. Sabouret, and S. Honiden. Tag Interactions in Multiagent Systems: Environment Support. In *Proceedings of the Second International Workshop on Environments for Multi-Agent Systems, Utrecht*, Lecture Notes in Computer Science, Vol. 3380. Springer Verlag, 2005.

[159] T. Prasad and D. Ok. Scaling Up Average Reward Reinforcement Learning by Approximating the Domain Models and the Value Function. In *Thirteenth International Conference on Machine Learning*, 1996.

[160] Z. Pylyshyn. *The Robot's Dilemma. The Frame Problem in Artificial Intelligence*. Ablex Publishing Corp., Norwood, New Jersey, 1987.

[161] L. Qiu, W. Hsu, S. Huang, and H. Wang. Scheduling and Routing Algorithms for AGVs: A Survey. *International Journal of Production Research*, 40(3), 2002.

[162] 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.

[163] A. Rao, M. Georgeff, and E. Sonenberg. Social Plans: A Preliminary Report. In *Decentralized AI 3, 3th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW*, Kaiserslautern, Germany, 1992. Elsevier Science B.V.: Amsterdam, Netherland.

[164] P. Reed. Reference Architecture: The Best of Best Practices. *The Rational Edge*, 2002. www-128.ibm.com/developerworks/rational/library/2774.html.

[165] C. Reynolds. Flocks, Herds and Schools: A Distributed Behavior Model. *Computer Graphics*, 21(4):25–34, 1996.

[166] J. Richter. *Applied Microsoft .NET Framework Programming.* Microsoft Press, Redmond, USA, 2002.

[167] G. Roman, C. Julien, and J. Payton. A Formal Treatment of Context-Awareness. *7th International Conference on Fundamental Approaches to Software Engineering*, 2004.

[168] J. Rosenblatt. DAMN: A Distributed Architecture for Mobile Navigation. In *Spring Symposium on Lessons Learned from Implemented Software Architectures for Physical Agents.* AAAI Press, 1995.

[169] K. Rosenblatt and D. Payton. *A Fine Grained Alternative to the Subsumption Architecture for Mobile Robot Control.* International Joint Conference on Neural Networks, IEEE, 1989.

[170] J. Rosenschein and L. Kaelbling. The Synthesis of Digital Machines With Provable Epistemic Properties. In *1st Conference on Theoretical Aspects of Reasoning about Knowledge, Monterey, CA*, 1986.

[171] N. Rozanski and E. Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives.* Addison Wesley Publishing Comp., 2005.

[172] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2003.

[173] J. Sauter and H. V. D. Parunak. *ANTS in the Supply Chain.* Agent based Decision Support for Managing the Internet-Enabled Supply Chain, Seattle, WA, 1999.

[174] 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.

[175] K. Schelfthout, T. Holvoet, and Y. Berbers. Views: Customizable Abstractions for Context-Aware Applications in MANETs. In *4th International Workshop on Software Engineering for Large-scale Multiagent Systems*, St. Louis, Missouri, 2005. ACM Press.

[176] K. Schelfthout, D. Weyns, and T. Holvoet. Middleware for Protocol-based Coordination in Dynamic Networks. In *3rd International Workshop on Middleware for Pervasive and Ad-hoc Computing*, Grenoble, France, 2005. ACM Press.

[177] 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.

[178] W. Schols. *Gradient Field Based Order Assignment in AGV Systems*. Master Thesis, Katholieke Universiteit Leuven, Belgium, 2005.

[179] 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/.

[180] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[181] O. Shehory. Architectural Properties of MultiAgent Systems. Technical Report CMU-RI-TR-98-28, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1998.

[182] M. Singh. Commitments Among Autonomous Agents in Information-Rich Environments. In *8th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, London, UK, 1997. Springer-Verlag.

[183] R. Smith. The Contract Net Protocol: High Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers*, 29(12):1104–1113, 1980.

[184] 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.

[185] 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.

[186] L. Steels. Cooperation between Distributed Agents through Self-Organization. *Decentralized Artificial Intelligence*, 1989.

[187] L. Steels. Exploiting Analogical Representations. *Designing Autonomous Agents, MIT Press*, 1990.

[188] K. Sycara. Multiagent Systems. *Artificial Intelligence*, 10(2):79–93, 1998.

[189] 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.

[190] F. Taghaboni and J. Tanchoco. Comparison of Dynamic Routing Techniques for Automated Guided Vehicle Systems. *International Journal of Production Research*, 33(10):2653–2669, 1995.

[191] B. Tekinerdogan. ASAAM: Aspectual Software Architecture Analysis Method. In *4th Working Conference on Software Architecture, WICSA, Oslo, Norway*. IEEE Computer Society, 2004.

[192] T. Tyrrell. *Computational Mechanisms for Action Selection*. PhD Dissertation, University of Edinburgh, 1993.

[193] P. Valckenaers and H. Van Brussel. Holonic Manufacturing Execution Systems. *CIRP Annals-Manufacturing Technology*, 54(1):427–432, 2005.

[194] T. Wagner and M. Pechouchek. *Proceedings of the Industry Track of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*. ACM, Hakodate, Japan, 2005.

[195] R. Want. System Challenges for Pervasive and Ubiquitous Computing (Intel). *Invited talk, International Conference on Software Engineeging, St. Louis, USA*, 2005.

[196] P. Wavish and D. Connah. Representing Multiagent Worlds in ABLE. *Technical Note, TN2964, Philips Research Laboratories*, 1990.

[197] E. Weisstein. Confidence Interval, Probability and Statistics, MathWorld. *http://mathworld.wolfram.com/ConfidenceInterval.html*, (6/2006).

[198] 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.

[199] D. Weyns, N. Boucké, T. Holvoet, and W. Schols. Gradient Field-Based Task Assignment in an AGV Transportation System. In *3th European Workshop on Multi-Agent Systems, Brussels, Belgium*. Koninklijke Vlaamse Academie van Belgie voor Wetenschappen en Kunsten, 2005.

[200] 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.

[201] D. Weyns and T. Holvoet. A Colored Petri Net for a Multi-Agent Application. In *Proceedings of Modeling Objects, Components and Agents, MOCA, University of Aarhus, Denmark*.

[202] D. Weyns and T. Holvoet. Look, Talk, and Do: A Synchronization Scheme for Situated Multiagent Systems. In *UK Workshop on Multi-Agent Systems*, Oxford, UK, 2002.

[203] 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.

[204] D. Weyns and T. Holvoet. Synchronous versus asynchronous collaboration in situated multi-agent systems. In *2nd International Joint Conference on Autonomous Agents and Multiagent Systems*, Melbourne, Australia, 2003. ACM.

[205] D. Weyns and T. Holvoet. A Colored Petri Net for Regional Synchronization in Situated Multiagent Systems. In *1st International Workshop on Coordination and Petri Nets, Bologna, Italy*, 2004.

[206] D. Weyns and T. Holvoet. Formal Model for Situated Multi-Agent Systems. *Fundamenta Informaticae*, 63(1-2):125–158, 2004.

[207] 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.

[208] D. Weyns and T. Holvoet. On Environments in Multiagent Systems. *AgentLink Newsletter*, 16:18–19, 2005.

[209] D. Weyns and T. Holvoet. On the Role of the Environment in Multiagent Systems. *Informatica*, 29(4):408–421, 2005.

[210] D. Weyns and T. Holvoet. A Reference Architecture for Situated Multiagent Systems. In *Environments for Multiagent Systems III, 3th International Workshop, E4MAS, Hakodate, Japan, 2006*, Lecture Notes in Computer Science. Springer, 2006.

[211] 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.

[212] D. Weyns and T. Holvoet. From Reactive Robotics to Situated Multiagent Systems: A Historical Perspective on the Role of Environment in Multiagent Systems. In *Engineering Societies in the Agents World VI, 6th International Workshop, ESAW, Kusadasi, Turkey*, Lecture Notes in Computer Science, Vol. 3963. Springer-Verlag, 2006.

[213] D. Weyns and T. Holvoet. Multiagent systems and Software Architecture. In *Special Track on Multiagent Systems and Software Architecture, Net.ObjectDays*, Erfurt, Germany, 2006.

[214] D. Weyns and T. Holvoet. Multiagent Systems and Software Architecture: Another Perspective on Software Engineering with Multiagent Systems. In *5th International Joint Conference on Autonomous Agents and Multi-Agent Systems, AAMAS, Hakodate, Japan*, 2006.

[215] D. Weyns, A. Omicini, and J. Odell. Environment as a First-Class Abstraction in Multiagent Systems. *Autonomous Agents and Multi-Agent Systems*, 14(1), 2007.

[216] D. Weyns, H. V. D. Parunak, F. Michel, T. Holvoet, and J. Ferber. Environments for Multiagent Systems, State-of-the-art and Research Challenges. Lecture Notes in Computer Science, Vol. 3374. Springer Verlag, 2005.

[217] D. Weyns, K. Schelfthout, and T. Holvoet. Architectural design of a distributed application with autonomic quality requirements. In *ICSE Workshop on design and evolution of autonomic application software, St. Louis, Missouri*, New York, NY, USA, 2005. ACM Press.

[218] D. Weyns, K. Schelfthout, and T. Holvoet. Exploiting a Virtual Environment in a Real-World Application. In *Proceedings of the Second International Workshop on Environments for Multi-Agent Systems, Utrecht*, Lecture Notes in Computer Science, Vol. 3830. Springer Verlag, 2005.

[219] D. Weyns, K. Schelfthout, T. Holvoet, and O. Glorieux. A Role Based Model for Adaptive Agents. In *4th Symposium on Adaptive Agents and Multi-Agent Systems*, UK, 2004.

[220] 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.

[221] 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.

[222] D. Weyns, K. Schelfthout, T. Holvoet, T. Lefever, and J. Wielemans. Architecture-centric development of an AGV transportation system. In *Multi-Agent Systems and Applications IV, 4th International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS, Budapest, Hungary*, Lecture Notes in Computer Science, Vol. 3690. Springer, 2005.

[223] D. Weyns, M. Schumacher, A. Ricci, M. Viroli, and T. Holvoet. Environments for Multiagent Systems. *Knowledge Engineering Review*, 20(2):127–141, 2005.

[224] D. Weyns, E. Steegmans, and T. Holvoet. A Model for Active Perception in Situated Multiagent Systems. In *1st European Workshop on Multi-Agent Systems*, Oxford, UK, 2003.

[225] D. Weyns, E. Steegmans, and T. Holvoet. Combining Adaptive Behavior and Role Modeling with Statecharts. In *3th International Workshop on Software Engineering for Large Scale Multiagent Systems*, Edingburg, Scotland, 2004.

[226] 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.

[227] 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.

[228] D. Weyns, E. Steegmans, and T. Holvoet. Towards Active Perception in Situated Multi-Agent Systems. *Applied Artificial Intelligence*, 18(9-10):867–883, 2004.

[229] D. Weyns, E. Steegmans, T. Holvoet, E. Helsen, and K. Deschacht. Delta Framework Cookbook. In *Technical Report*. Departement of Computer Science, Katholieke Universiteit Leuven, Belgium. http://www.cs.kuleuven.ac.be/publicaties/rapporten/CW/2005/, 2006.

[230] D. Weyns, G. Vizzari, and T. Holvoet. Environments for situated multiagent systems: Beyond Infrastructure. In *Proceedings of the Second International Workshop on Environments for Multi-Agent Systems, Utrecht, 2005*, Lecture Notes in Computer Science, Vol. 3380. Springer Verlag.

[231] S. Whiteson and P. Stone. Adaptive Job Routing and Scheduling. *Engineering Applications of Artificial Intelligence Special Issue on Autonomic Computing and Automation*, 17(7):855–869, 2004.

[232] T. De Wolf, G. Samaey, T. Holvoet, and D. Roose. Decentralised Autonomic Computing: Analysing Self-Organising Emergent Behaviour Using Advanced Numerical Methods. In *2nd International Conference on Autonomic Computing, IEEE Computer Society*, 2005.

[233] M. Wood and S. DeLoach. An Overview of the Multiagent Systems Engineering Methodology. In *Agent-Oriented Software Engineering I*, Volume 1957 of Lecture Notes in Computer Science, Vol. 1957. Springer-Verlag, New York, 2000.

[234] S. Woods and M. Barbacci. Architectural Evaluation of Collaborative Agent-Based Systems. Technical Report CMU/SEI-99-TR-025, Software Engineering Institute, Carnegie Mellon University, PA, USA, 1999.

[235] M. Wooldridge, N. Jennings, and D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[236] 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.

[237] L. Xu and H. Weigand. The Evolution of the Contract Net Protocol. In *Advances in Web Age Information Systems, 2nd Internation Conference*, Lecture Notes in Computer Science, Vol. 2118. Springer, 2001.

[238] D. Yamins. Towards a Theory of 'Local to Global' in Distributed Multiagent Systems. In *4th Joint Conference on Autonomous Agents and Multiagent Systems*, Utrecht, The Netherlands, 2005.

[239] E. Yu. *Modelling Strategic Relationships for Process Reengineering.* 1995. PhD Dissertation: University of Toronto, Canada.

[240] F. Zambonelli, N. Jennings, and M. Wooldridge. Developing Multiagent Systems: The Gaia Methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):317–370, 2003.

[241] F. Zambonelli and A. Omicini. Challenges and Research Directions in Agent-Oriented Software Engineering. *Journal of Autonomous Agents and Multiagent Systems*, 9(3):253–283, 2003.

[242] F. Zambonelli and H. V. D. Parunak. *From Design to Intention: Signs of a Revolution*. 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems, Bologna, Italy, ACM Press, New York, 2002.

[243] K. Zeghal and J. Ferber. CRAASH: A Coordinated Collision Avoidance System. In *European Simulation Conference*, Lyon, France, 1993.

Appendix A

Formal Specification of the Reference Architecture

This appendix gives a formal specification of the reference architecture for situated multiagent systems. The specification precisely describes the core properties of the architectural elements and it defines the constraints the architectural elements have to comply to. We employ a simple formal notation based on set theory. The specification gradually introduces new concepts to specify the properties of architectural elements in subsequent views. The formalization is delimitated by the specification of concepts that introduce variability of the reference architecture. These concepts are not further specified, but, when applying the reference architecture, these concepts have to be defined according to the requirements of the application at hand. Not further specified concepts typically relate to variation mechanisms that are described in the reference architecture documentation.

The specification consists of three parts that correspond to the first three views of the reference architecture described in chapter 4. Part A.1 specifies the architectural elements of the module view of the reference architecture (described in section 4.3). Part A.2 specifies the architectural elements of the shared data view (section 4.4). Finally, part A.3 specifies the elements of the collaborating components view (section 4.5). The specification of the architectural elements of a view is divided in view packets that correspond to the view packets of the reference architecture documentation. In each view packet, we first retake the overview of the architectural elements from the architecture documentation. For the view packets of the collaborating components view, the overviews are annotated with symbols used in the formal specification. Then follows the specification of the main properties of the architectural elements and the relationships between the elements.

A.1 Architecture Elements of the Module Decomposition View

The specification of the architectural elements of the module decomposition view is structured according to the three view packets: the top-level decomposition of the situated multiagent system, the primary decomposition of an agent, and the primary decomposition of the application environment.

A.1.1 Module Decomposition View Packet 1: Situated Multiagent System

A.1.1.1 Architectural Elements and Relationships

Fig. A.1 shows the top-level decomposition of a situated multiagent system with the deployment context.

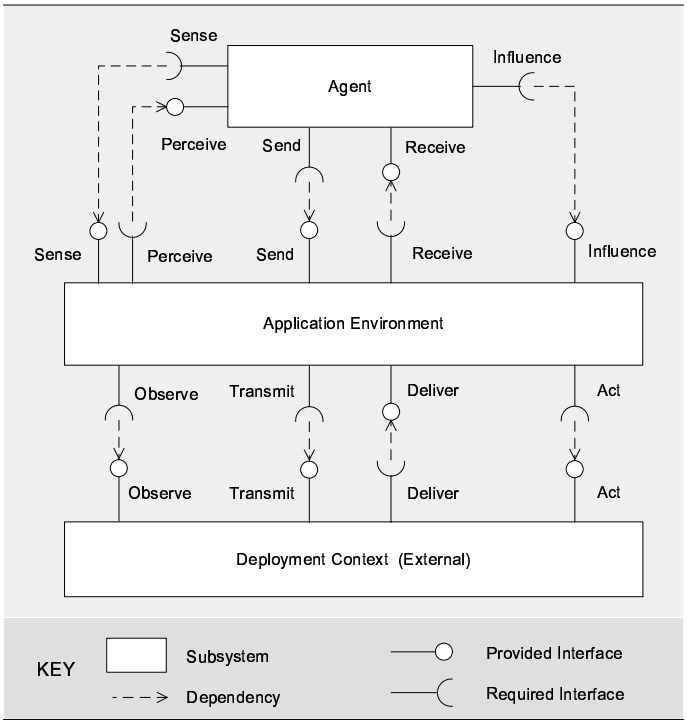


Figure A.1: Top-level decomposition of a situated multiagent system with the deployment context

A.1.1.2 Specification of Architectural Elements

General Definitions

Ag	$Ag = \{a_1, .., a_i, .., a_n\}$ is the set of agents in the multiagent system; the index $i \in \{1, .., n\}$ is a unique identifier for agent $a_i \in Ag$; we use $Y = \{1, .., n\}$ to denote the set of agent identifiers in the system
P_{Ag}	a set partition of Ag; $P_{Ag} = \{Ag1, .., Agt\}$ is a disjoint subsets of agents; each subset $Agi \in P_{Ag}$ groups the agents of one type; agents of a type are assigned the same application goals and have the same architecture structures; for the application-specific definition of agent types, see variation mechanism M1 (section 4.3.1.4)
Ont	the ontology that defines the terminology of the application domain; Ont is defined as a tuple $\langle Voc, Rel \rangle$ with: 1. Voc: the vocabulary of domain concepts 2. Rel: the set of relationships between concepts of Voc Voc and Rel are not further specified, for the application-specific definition of Ont, see variation mechanism M2 (section 4.3.1.4)

Agent – Application Environment Interface

`Sense and Perceive.` The `Sense` interface enables an agent to sense the environment selectively. The application environment requires the interface `Perceive` to pass on the resulting representation to the agent.

Fo	the set of foci for agents to sense the environment selectively, a focus $fo \in Fo$ is defined as a 3-tuple $\langle i, foname, foparam \rangle$ with $i \in Y$ the identity of the agent, $foname$ a name that refers to the type of information the agent aims to observe, and $foparam$ a set of additional scoping parameters of the focus; $foname$ and $foparam$ are not further specified, see observation mechanism M5 (section 4.3.2.4)
$\Theta \subseteq 2^{Fo}$	the powerset of foci in the agent system (the power set of a set S, denoted as 2^S, is the set of all subsets of S)
R	the set of representations of the environment, a representation $r \in R$ is a data structure that refers back to elements or resources in the environment; representations are not further specified, see variation mechanism M6 (section 4.3.2.4)

Interface definition `Sense`

$void : sense(\Theta)$	senses the environment with a given set of foci

Interface definition `Perceive`

$void : perceive(R)$	perceives the given representation

`Send and Receive.` The `Send` interface enables an agent to send messages to other agents. The application environment requires the interface `Receive` to deliver messages to agents.

M	the set of messages that can be sent by agents in the system; a message $m_{i \to des} \in M$ is a formatted structure of characters that represents a message sent by the agent with identity $i \in Y$ to a set of agents with identities specified in $des \in 2^Y$; for brevity we use $m_{i,j}$ to denote a message sent by the agent with identity $i \in Y$—the sender, to the agent with identity $j \in Y$—the addressee
Ont'	the ontology that defines a shared vocabulary of words that agents use to represent domain concepts and relationships between the concepts in messages; Ont' is defined as a tuple $\langle Voc', Rel' \rangle$ with Voc' a vocabulary of domain concepts and Rel' a set of relationships of the ontology; Ont' is typically a part of the integral ontology Ont of the application domain (see section A.1.1).
L	the communication language that defines the format of messages; a message is a 5-tuple with the following fields: 1. $cid \in Cid$: a unique id of the conversation, with Cid the set of conversation ids; the function $Cid()$ returns a new id to the initiator of a conversation 2. $sender \in Y$: the id of the sender of the message 3. $addressees \in 2^Y$: the ids of the addressees 4. $Perf$: the set of performatives of L 5. $Cont^{Ont'}$: the set of contents of L based on the ontology Ont' Cid, $Perf$ and $Cont^{Ont'}$ are not further specified, for the domain-specific definition of the communication language, see variation mechanism M10 (section 4.3.2.4)

Interface definition `Send`

$void : send(M)$	sends the given message to the addressees indicated in the message

Interface definition `Receive`

$void : receive(M)$	receives the given message; the addressee is denoted in the message

`Influence.` The application environment provides the interface `Influence` that enables agents to invoke influences in the environment.

Inf	the set of influences that can be invoked in the environment; an influence $inf \in Inf$ is defined as a 3-tuple $\langle i, iname, iparam \rangle$ with $i \in Y$ the identity of the agent, $iname$ a name that refers to the type of influence the agent invokes and $iparam$ is a set of additional parameters of the influence; $iname$ and $iparam$ are not further specified, for the application-specific definition of influences, see variation mechanism M8 (section 4.3.2.4)

Interface definition `Influence`

$void : influence(Inf)$	invokes the given influence in the environment

Application Environment – Deployment Context Interface

`Observe.` The application environment requires the `Observe` interface to observe the state of external resources in the deployment context.

O_{DC}	the set of observation primitives to observe the deployment context; an observation primitive is a 3-tuple $\langle obpres, obptype, obpparam \rangle$; $obpres$ refers to the resource that is subject of observation, $obptype$ describes the type of information that is observed, and $obpparam$ is a set of additional parameters of the observation primitive; $obpres$, $obptype$, and $obpparam$ are not further specified, see variation mechanism M3 (section 4.3.1.4)
S_{DC}	the set of resource data observable from the deployment context; a resource data is a data structure that refers back to resources in the deployment context; resource data are not further specified, see variation mechanism M3 (section 4.3.1.4)
$\Omega \subseteq 2^{S_{DC}}$	the power set of resource data that can be observed from the deployment context

Interface definition `Observe`

$\Omega : observe(O_{DC})$	collects a set of resource data from the deployment context with a given observation primitive

For convenience, we have specified *observe* as a synchronous operation. Yet, in practice, alternative mechanisms such as publish/subscribe and callback can be used to observe the deployment context.

`Transmit and Deliver.` The application environment requires the interface `Transmit` to send messages to agents. The deployment context requires the interface `Deliver` from the application environment to deliver the incoming messages to the agents.

M_{DC}	the set of low-level formatted messages that can be transmitted via the deployment context; a low-level formatted message is a structured set of characters that represents a message exchanged between a sender and one or more addressees; low-level formatted messages are not further specified, see variation mechanism M3 (section 4.3.1.4)

Interface definition `Transmit`

$void : transmit(M_{DC})$	transmits the given low-level formatted message to the addressees indicated in the message

Interface definition `Deliver`

$void : deliver(M_{DC})$	delivers the given low-level formatted message to the addressee indicated in the message

Operations $transmit(M_{DC})$ and $deliver(M_{DC})$ provide a message transfer service to transmit low-level formatted messages via the deployment context.

`Act.` The application environment requires the interface `Act` to modify the state of external resources (based on influences invoked by agents).

A_{DC}	the set of action primitives to access the deployment context; an action primitive is a 3-tuple $\langle actres, actname, actparam \rangle$; $actres$ refers to the target resource, $actname$ describes the type of action, and $actparam$ is a set of additional parameters of the action primitive; $actres$, $actname$, and $actparam$ are not further specified, see variation mechanism M3 (section 4.3.1.4)

Interface definition `Act`

$void : act(A_{DC})$	invokes the given low-level action primitive in the deployment context

A.1.2 Module Decomposition View Packet 2: Agent

A.1.2.1 Architectural Elements and Relationships

Fig. A.2 shows the module decomposition of an agent.

A.1.2.2 Specification of Architectural Elements

Perception – Communication/Decision Making Interface

`Request.` The provided `Request` interface of the perception module enables decision making and communication to request a perception of the environment.

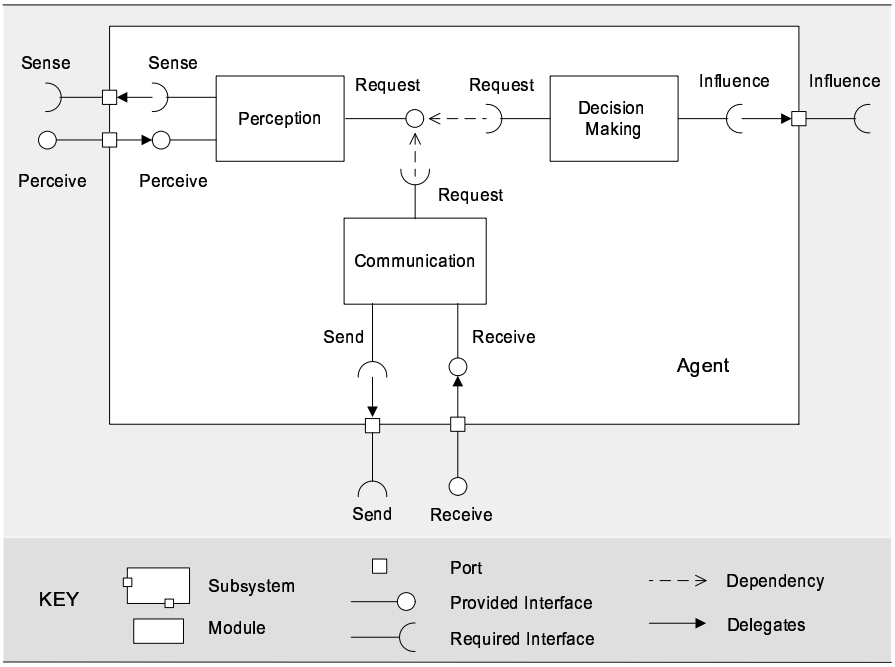


Figure A.2: Module decomposition of an agent

$Fo_i \subseteq Fo$	the set of foci available to agent a_i
$\Theta_i \subseteq 2^{Fo_i}$	the set of focus combinations of agent a_i; this set contains all focus combinations that can be selected by agent a_i
S_Θ	the set of focus selectors in the agent system; a focus selector $s_\theta \in S_\Theta$ specifies a focus combination $\theta \in \Theta$, and is defined as $\{fos_1, .., fos_s\}$ with fos_p defined as $\langle\, fosname_p, fosparam_p \,\rangle$ with $fosname_p$ the name of a selected focus and $fosparam_p$ a set of additional parameters of that focus; focus selectors are not further specified, see variation mechanism M5 (section 4.3.2.4)
S_{Θ_i}	the set of focus selectors available to agent a_i
K_{Ag}^{Ont}	the set of state elements of agents of Ag based on the ontology Ont; a state element represents knowledge of an agent about the application domain and can refer to elements external to the agent as well as state private to the agent; a state element is defined as a tuple $\langle\, kname, kfields \,\rangle$; $kname$ is the name of the state element and $kfields$ a set of fields, each field consisting of a name and a value of an accompanying domain; $kname$ and $kfields$ are not further specified, for the application-specific definition of K_{Ag}^{Ont} see variation mechanism SD1 (section 4.4.1.4)

$K_i \subseteq K_{Ag}^{Ont}$	the set of all state elements of agent $a_i \in Ag$
$\mathcal{K}_i \subseteq 2^{K_i}$	the power set of state elements of agent a_i; we denote the actual set of state elements $\kappa \in \mathcal{K}_i$ as the *current knowledge* of the agent
$P_i \subseteq \mathcal{K}_i$	the set of percepts of agent a_i; a percept $p \in P_i$ represents knowledge of agent a_i about the environment
Fi_i	the set of filters of agent a_i; a filter $fi \in Fi_i$ is typed as $fi : p \rightarrow Bool$ with $p \in P_i$ and $Bool = \{true, false\}$; i.e. a filter maps state elements of a percept onto boolean values; all state elements $s \in p$ for which $fi(s)$ returns *true* can pass the filter, the elements that return $false$ are blocked; filters are not further specified, see variation mechanism M7 (section 4.3.2.4)
$\Phi_i \subseteq 2^{Fi_i}$	the set of filter combinations of agent a_i; this set contains all filter combinations that can be selected by agent a_i to filter percepts
S_{Φ_i}	the set of filter selectors available to agent a_i; a filter selector $s_\phi \in S_{\Phi_i}$ specifies a filter combination $\phi \in \Phi_i$, and is defined as $\{fis_1, .., fis_r\}$ with fis_q defined as $\langle\, fisname_q, fisparam_q \,\rangle$ with $fisname_q$ the name of a selected filter and $fisparam_q$ a set of additional parameters of that filter; filter selectors are not further specified, see variation mechanism M7 (section 4.3.2.4)

Interface definition `Request`

$void : request(S_{\Theta_i}, S_{\Phi_i})$	requests a perception with the given focus and filter selectors

A.1.3 Module Decomposition View Packet 3: Application Environment

A.1.3.1 Architectural Elements and Relationships

Fig. A.3 shows the module decomposition of the application environment.

A.1.3.2 Specification of Architectural Elements

Perception Generator – Observation & Data Processing Interface

`Collect and Generate.` To observe resources in the deployment context, the perception generator depends on the `Collect` interface of the observation & data processing module. The observation & data processing module uses the `Generate` interface provided by the perception generator to generate a representation based on the data derived from the observed resources.

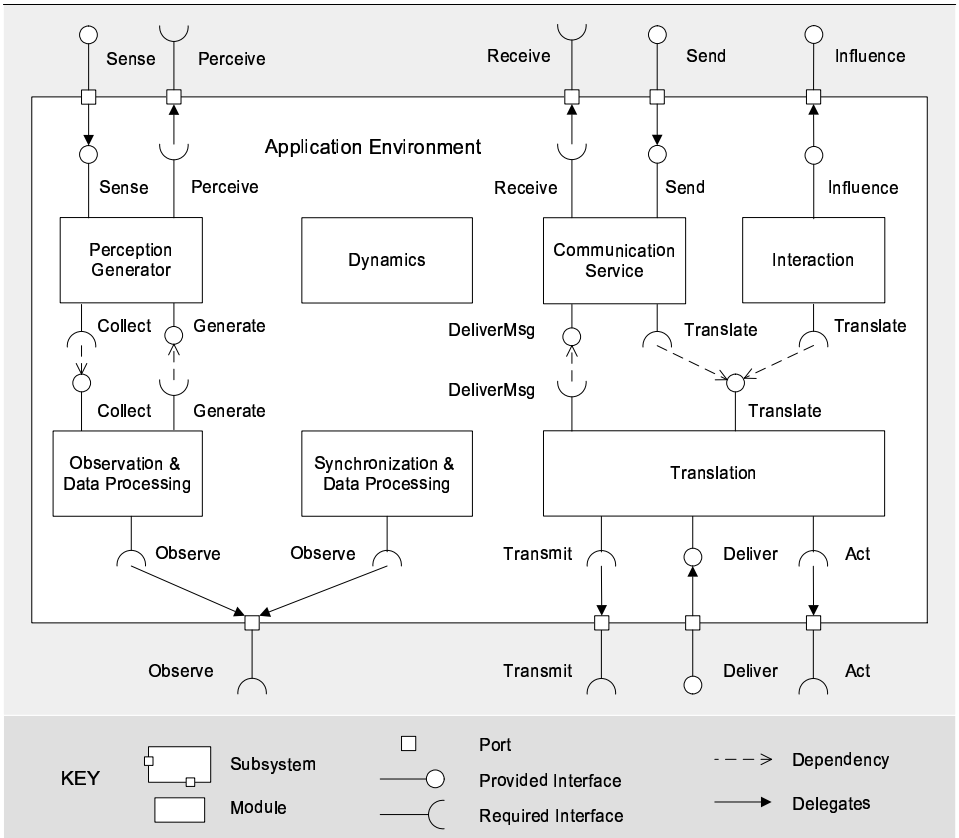


Figure A.3: Module decomposition of the application environment

S_E^{Ont}	the set of state elements of the application environment based on ontology Ont; a state element of the application environment represents a part of the state of the application environment or the deployment context and is defined as $\langle sname, sfields \rangle$ with $sname$ the name of the state element, and $sfields$ a set of fields, each field consisting of a name and a value of an accompanying domain; $sname$ and $sfields$ are not further specified, for the application-specific definition of S_E^{Ont} see variation mechanism SD2 (section 4.4.2.4) (for brevity, we use S_E instead of S_E^{Ont} hereafter)
$S_{E\mathcal{A}}$	$S_{E\mathcal{A}} \subseteq S_E$ is the set of state elements that refer to elements of the application environment
$S_{E\mathcal{D}}$	$S_{E\mathcal{D}} \subseteq S_E$ is the set of state elements that can be deduced from the observation of the deployment context

$\Psi \subseteq 2^{S_{E\mathcal{D}}}$	the powerset of state elements of the application environment that can be deduced from the observation of the deployment context
O	the set of observations to collect data from the observation of the deployment context; an observation $ob \in O$ is typed as follows: $ob : S_{E\mathcal{D}} \rightarrow Bool$; i.e. an observation is a function that maps state elements of $S_{E\mathcal{D}}$ on boolean values; $ob(s)$ returns $true$ for state elements $s \in S_{E\mathcal{D}}$ that are targeted by the observation, and $false$ otherwise; see variation mechanism M14 (section 4.3.3.4)

Interface definition `Collect`

$void : collect(O)$	collects data from the deployment context with the given observation

Interface definition `Generate`

$void : generate(\Psi)$	generates a representation with the given state elements collected from the observation of the deployment context

Communication Service/Interaction – Translation Interface

`DeliverMsg.` Translation converts incoming messages into an appropriate format for agents and uses the `DeliverMsg` interface of the communication service to deliver the messages.

Interface definition `DeliverMsg`

$void : deliver(M)$	delivers the given message to the addressee indicated in the message

`Translate.` The `Translate` interface of the translation module provides a dual functionality: (1) it converts messages into a low-level format for transmission via the deployment context, and (2) it converts operations (from the influences invoked by agents) into low-level action primitives of the deployment context.

G	the set of operations in the agent system; an operation $g \in G$ is typed as follows: $g : S_E \rightarrow \{ true, false, unspec \}$; $g(st) = true$ denotes that $st \in S_E$ is part of the target state of the operation, $g(sf) = false$ denotes that $sf \in S_E$ is *not* part of the target state, and finally $g(su) = unspec$ denotes that $su \in S_E$ is invariable to the operation; for the application-specific definition of operations, see variation mechanism M14 (section 4.3.3.4)

Interface definition `Translate` provided by Translation

$void : translate(M)$	translates the given message to a low-level interaction with the deployment context

$void : translate(G)$	translates the given operation to a low-level interaction with the deployment context

A.2 Architecture Elements of the Shared Data View

The specification of the architectural elements of the shared data view is structured according to the two view packets of the view: situated agent and the application environment.

A.2.1 C & C Shared Data View Packet 1: Agent

A.2.1.1 Architectural Elements and Relationships

Fig. A.4 shows the shared data view of a situated agent.

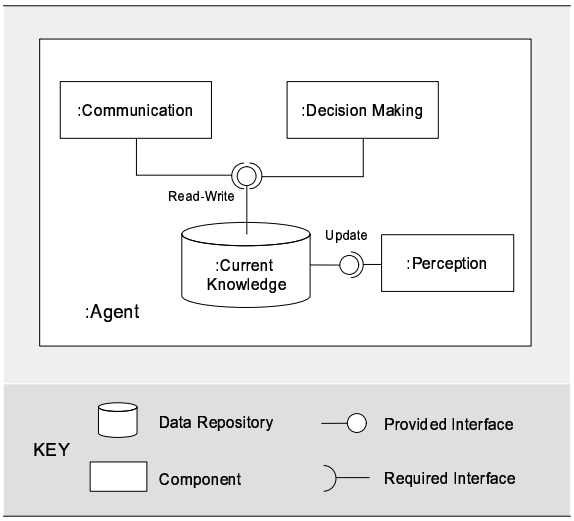


Figure A.4: Repository and data accessors of an agent

A.2.1.2 Specification of Architectural Elements

Current Knowledge – Communication/Decision Making Interface

`Read-Write`. The `Read-Write` interface enables the communication and decision making component to read and modify the agent's current knowledge.

Interface definition `Read-Write`

$\mathcal{K}_i : read()$	reads the agent's current knowledge
$void : write(2^{K_i})$	writes the given set of state elements in the agent's current knowledge

Current Knowledge – Perception Interface

`Update.` The `Update` interface enables the perception component to update the agents knowledge according to the information derived from sensing the environment.

Interface definition `Update`

$void : update(P_i)$	updates the agent's current knowledge with the given percept

A.2.2 C & C Shared Data View Packet 2: Application Environment

A.2.2.1 Architectural Elements and Relationships

Fig. A.5 shows the shared data view of the application environment.

A.2.2.2 Specification of Architectural Elements

Interfaces Provided by State

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

$\Sigma \subseteq 2^{S_E}$	the powerset of state element of the State repository; we denote the actual set of state elements $\sigma \in \Sigma$ as the *current state* of the application environment

Interface definition `Read`

$\Sigma : read()$	reads the current state of the application environment from the state repository

Interface definition `Read-Write`

$\Sigma : read()$	reads the current state of the application environment
$void : write(2^{S_E})$	writes the give state elements in the state repository of the application environment; overwrites old state if applicable

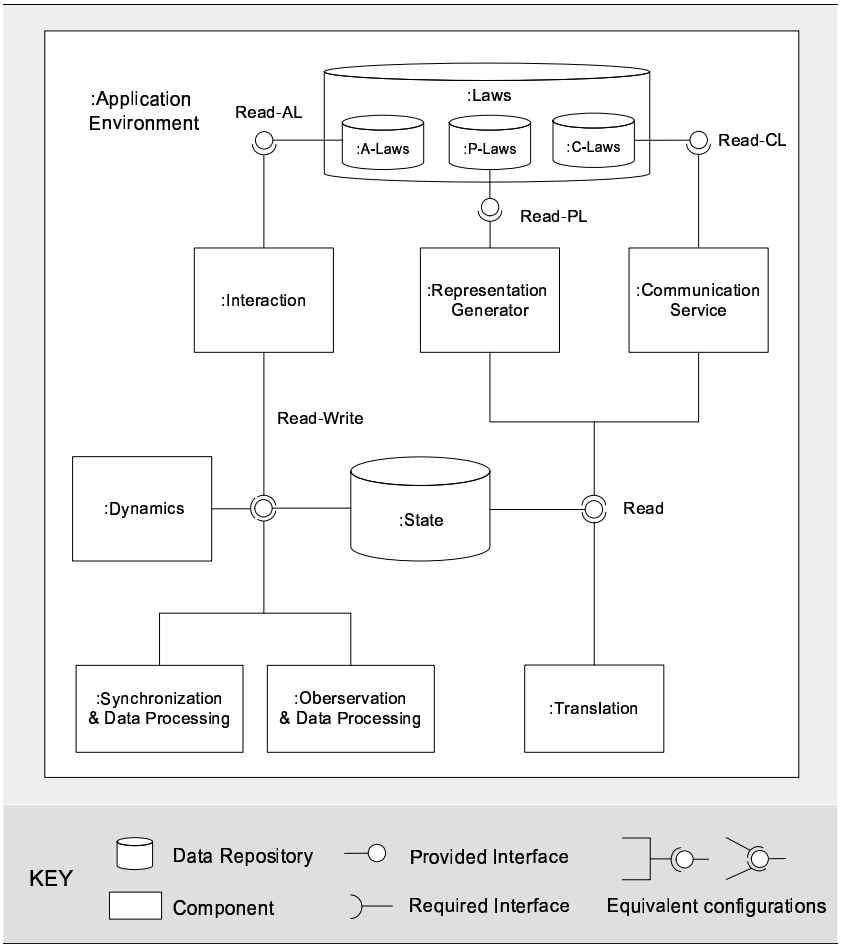


Figure A.5: Repositories and data accessors of the application environment

Interfaces Provided by Laws

`Read-PL`. The `Read-PL` interface provided by the Laws repository enables the representation generator to consult the perception laws.

Sc	the set of perception scopes (or scopes for short); a scope $sc \in Sc$ is typed as $sc : S_E \rightarrow Bool$; i.e. a scope maps state elements of the environment on booleans; $sc(si)$ returns $true$ for the elements $si \in S_E$ that are within the scope of sc, and $sc(so) = false$ for the elements $so \in S_E$ outside the scope; we call the set of state elements that map on true as the *domain of interest* of a scope

Co_P	the set of perception constraints in the agent system; a perception constraint $cop \in Co_P$ is typed as $cop : S_E \rightarrow Bool$; i.e. a perception constraint maps state elements $s \in S_E$ on booleans restricting agents' perception of the environment; $cop(s)$ returns $true$ for the state elements $s \in S_E$ that are restricted for perception, and $false$ for unconstrained state elements
L_P	the set of perception laws; a perception law $lp \in L_P$ is typed as $lp : Sc \times \Sigma \rightarrow Co_P$; i.e. a perception law $lp(sc, \sigma) = cop$ takes a scope sc together with the current state of the application environment σ and produces a perception constraint cop

Interface definition `Read-PL`

$L_P : plaws()$	consults the perception laws

`Read-AL`. The `Read-AL` interface provided by the Laws repository enables the interaction component to consult the action laws.

Co_A	the set of operation constraints in the agent system; an operation constraint $coa \in Co_A$ is typed as $coa : S_E \rightarrow Bool$; i.e. an operation constraint maps state elements $s \in S_E$ on booleans restricting agents' actions in the environment; $coa(s)$ returns $true$ for the state elements that are constrained for modification, and $false$ for unconstrained state elements
L_A	the set of action laws in the agent system; an action law $la \in L_A$ is typed as follows: $la : G \times \Sigma \rightarrow Co_A$; an action law $la(g, \sigma) = coa$ takes an operation g and the current state of the environment σ and returns an operation constraint coa

Interface definition `Read-AL`

$L_A : alaws()$	consults the action laws

`Read-CL`. The `Read-CL` interface provided by the Laws repository enables the communication service to consult the communication laws.

Co_C	the set of communication constraints in the multiagent system; a communication constraint $coc \in Co_C$ is typed as $coc : Y \rightarrow Bool$; i.e. a communication constraint maps identities of agents $i \in Y$ on booleans; coc(i) returns true for identities of agents that are excluded for a particular message, and false for non constrained identities
L_C	the set of communication laws in the agent system; a communication law $lc \in L_C$ is typed as follows: $lc : M \times \Sigma \rightarrow Co_C$; a communication law $lc(m_{i \rightarrow des}, \sigma) = coc$ takes a message $m_{i \rightarrow des}$ and the current state of the environment σ and returns a communication constraint coc

Interface definition `Read-CL`

$L_C : claws()$	consults the communication laws

A.3 Architecture Elements of the Collaborating Components View

The specification of the architectural elements of the collaborating components view is structured according to the three view packets of the view. Subsequently, we discuss the view packet with the collaborating components of perception, interaction, and communication. Each view packet starts with the specification of a number of additional concepts. Then the various collaboration components are specified.

A.3.1 C & C Collaborating Components View Packet 1: Perception and Representation Generator

A.3.1.1 Architectural Elements and Relationships

Fig. A.6 shows the collaborating components that realize the functionality for perception.

A.3.1.2 Specification of Architectural Elements

General Definitions

X_i	the set of descriptions of agent a_i; a description $\chi \in X_i$ is typed as: $\chi : R \rightarrow 2^{K_i}$; i.e. a description maps a representation $r \in R$ on a set of state elements of 2^{K_i}; for the application-specific definition of descriptions, see variation mechanism CC1 (section 4.5.1.3)

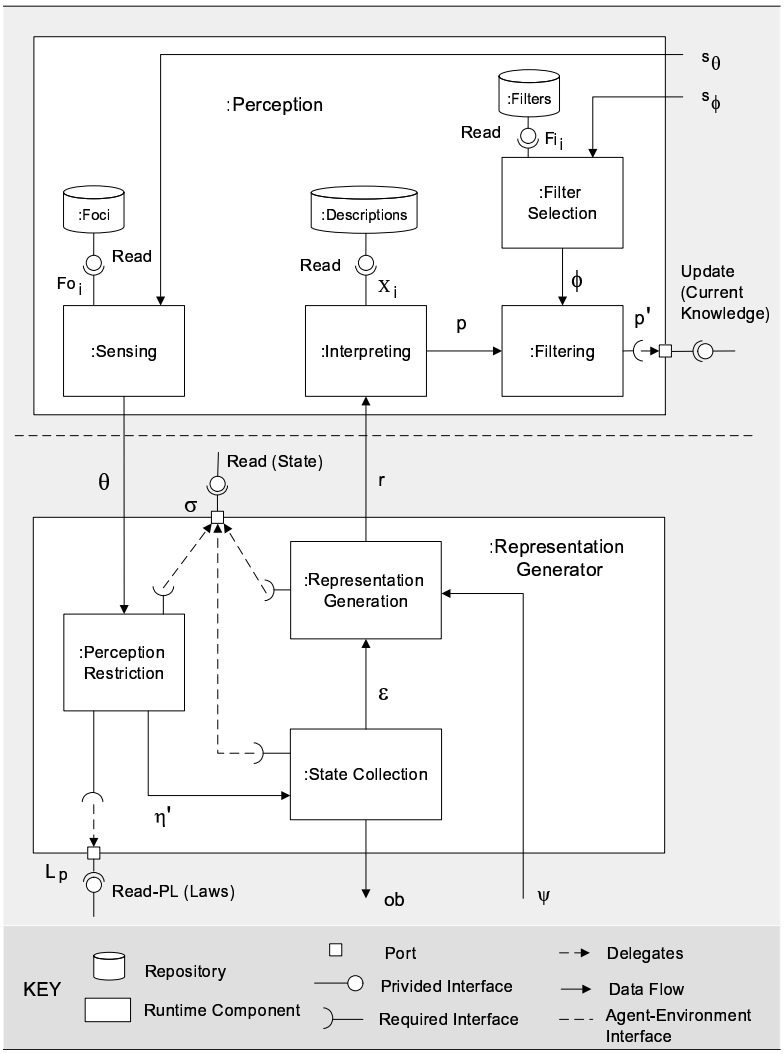


Figure A.6: Collaborating components of perception and representation generator

$\mathcal{N} \subseteq 2^{Sc}$	the powerset of scopes in the agent system; $\eta \in \mathcal{N}$ is a set of scopes derived from a set of foci of a perception request; the convertion of foci to scopes is the responsibility of the *Scoping* component (see the definition of the collaborating components below)

$\mathcal{E} \subseteq \Sigma$	the set of state element sets that can be observed from the state of the application environment;
$dom(f)$	we use the following notation: for a function $f : D \rightarrow \{v1, .., vn\}$, we use $dom(f)$ to denote the domain of f, thus $dom(f) = D$; we use $dom(f)^{\rightarrow vi}$ to denote the subdomain of elements of $dom(f)$ that map to vi, with $vi \in \{v1, .., vn\}$

Collaborating Components

Components of agents are specified per agent, indicated with a subscript i. In practice, agents of a particular agent type usually share the same component definition (see section A.1.1).

$Sensing_i$ takes a focus selector and the set of foci of the agent[1], and selects the corresponding foci to produce a perception request. Sensing is typed as follows:

$$Sensing_i : S_{\Theta_i} \rightarrow \Theta_i$$
$$Sensing_i(s_\theta) = \theta$$

PerceptionRestriction takes the set of foci from a perception request, the current state of the application environment, and generates according to the set of perception laws, a perception scope for the perception request. PerceptionRestriction is typed as follows:

$$PerceptionRestriction : \Theta \times \Sigma \rightarrow \mathcal{N}$$
$$PerceptionRestriction(\theta, \sigma) = \eta'$$

PerceptionRestriction first converts the set of foci of a perception request into a set of scopes, and subsequently, it applies the perception laws to this set of scopes. The conversion of foci to scopes is defined by the *Scoping* function that is typed as follows:

$$Scoping : \Theta \rightarrow \mathcal{N}$$
$$Scoping(\theta) = \eta$$

The application of the perception laws is defined by the $ApplyL_P$ function that is typed as follows:

$$ApplyL_P : \mathcal{N} \times \Sigma \rightarrow \mathcal{N};$$
$$ApplyL_P(\eta, \sigma) = \eta'$$

[1] Given sets for a particular agent, such as the sets of foci of filters, are not explicitly indicated in the definition of the components. In a similar way, given sets of the application environment, such as the various sets of laws, are not explicitly indicated in component definitions of the application environment.

$ApplyL_P$ applies the set of perception laws L_p to a set of scopes η, given the current state of the environment σ. $ApplyL_P$ results in a restricted perception scope η'. For the resulting perception scope holds:

$$\forall s \in S_E : \eta'(s) = true \text{ iff } (\exists\ sct \in \eta : sct(s) = true)\ \wedge \\ (\forall\ lp \in L_P, \forall\ sc \in \eta : (\forall\ co \in lp\,(sc, \sigma) : co(s) = false))) \\ \text{otherwise } \eta'(s) = false$$

That is, a state element is within the restricted perception scope if (i) the state element is within the domain of interest of at least one scope, and (ii) none of the constraints of the applied perception laws is applicable to the state element.

For the resulting perception scope holds:

$$dom(\eta')^{\rightarrow true} = (\bigcup_{sc \in \eta} dom(sc)^{\rightarrow true}) \\ \bigcap\ (\bigcup_{lp \in L_P,\ sc \in \eta,\ co \in lp(sc,\sigma)} dom(co)^{\rightarrow false})$$

The observable domain of the perception scope (i.e. the subdomain of elements of observable state of the environment that map to true) consists of the intersection of the domain of interest of the scopes of the perception request and the subdomain of elements of the state of the environment that are not constrained by the perception laws.

StateCollection collects the observable state for a perception request, given the current state of the application environment. StateCollection selects the subset of state elements of the application environment for the given perception scope, and produces an observation to collect data from the deployment context within perception scope. StateCollection is typed as follows:

$$StateCollection : \mathcal{N} \times \Sigma \rightarrow \mathcal{E} \times O$$
$$StateCollection(\eta', \sigma) = (\varepsilon, ob)$$

RepresentationGeneration takes the observed state of a perception request and produces a representation. RepresentationGeneration is typed as follows:

$$RepresentationGeneration : \mathcal{E} \times \Psi \times \Sigma \rightarrow R$$
$$RepresentationGeneration(\varepsilon, \psi, \sigma) = r$$

The generated representation integrates the state observed from the application environment ε with the state collected from the observation of the deployment context ψ.

$Interpreting_i$ uses the set of descriptions to interpret a given representation resulting in a percept for the agent. Interpreting is typed as follows:

$$Interpreting_i : R \rightarrow P_i$$
$$Interpreting_i(r) = p$$

$FilterSelection_i$ selects a subset of filters from the agent's set of filters according to a given filter selector.

$$FilterSelection_i : S_{\Phi_i} \rightarrow \Phi_i$$
$$FilterSelection_i(s_\phi) = \phi$$

$Filtering_i$ finally, filters a percept of an agent according to the set of selected filters. Filtering is typed as follows:

$$Filtering_i : P_i \times \Phi_i \rightarrow P_i$$
$$Filtering_i(p, \phi) = p'$$

For the elements of the given percept p that have passed the set of filters ϕ holds:

$$\forall k \in p' : (k \in p) \ \wedge (\nexists fi \in \phi_i : fi(k) = false)$$

This expression states that the filtered percept only includes the elements of the original percept for which no filter of the set of selected filters blocks the elements.

A.3.2 C & C Collaborating Components View Packet 2: Decision Making and Interaction

A.3.2.1 Architectural Elements and Relationships

Fig. A.7 shows the collaborating components that realize the functionality for actions.

A.3.2.2 Specification of Architectural Elements

General Definitions

T	the set of stimuli in the agent system; a stimulus is factor internal to the agent, or a factor that refers to external elements in the environment, that drives an agent's decision making process; a stimulus $t \in T$ is defined as $\langle stname, stfields \rangle$ with $stname$ the name of the stimulus, and $stfields$ a set of fields, each field consisting of a name and a value of an accompanying domain; $stname$ and $stfields$ are not further specified; the application-specific definition of stimuli depends on the applied mechanism for action selection, see variation mechanism CC3 (section 4.5.2.3)
$T_i \subseteq T$	the set of stimuli of agent a_i
$\mathcal{T}_i = 2^{T_i}$	the powerset of stimuli of agent a_i

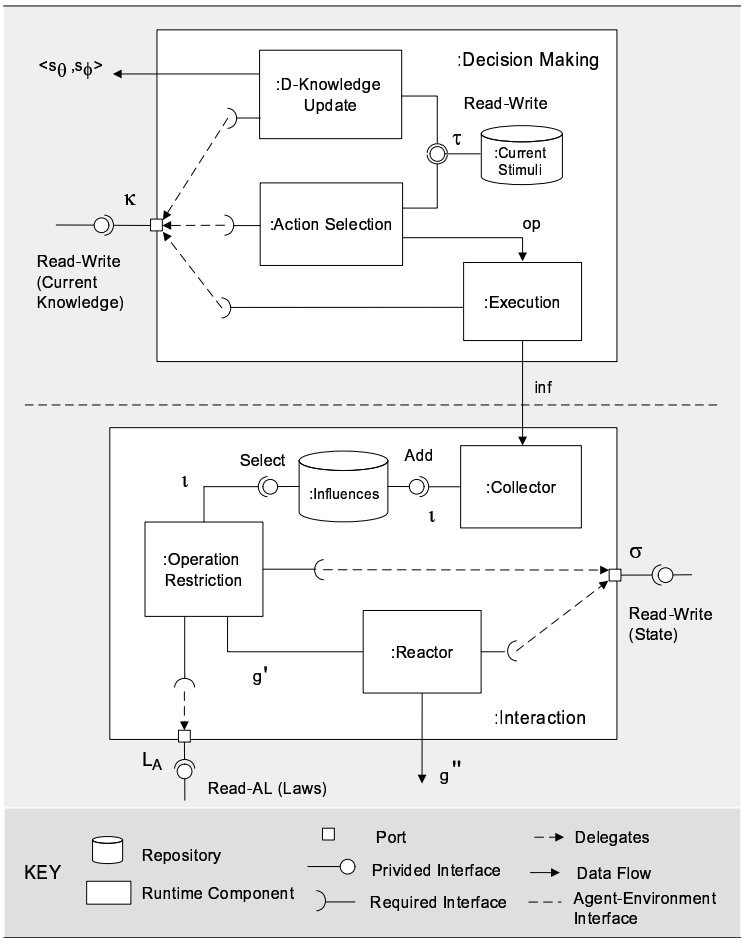


Figure A.7: Collaborating components of decision making and interaction

Op	the set of operators in the agent system; an operator $op \in Op$ is defined as a tuple $\langle opname, opparam \rangle$ with $opname$ a name that refers to the type of operator, and $opparam$ a set of additional parameters of the operator; $opname$ and $opparam$ are not further specified, see variation mechanism CC3 (section 4.5.2.3)
$Op_i \subseteq Op$	the set of operators that can be selected by agent a_i
$\mathcal{I} = 2^{Inf}$	the powerset of influences; $\iota \in \mathcal{I}$ is a set of influences in the agent system
$\Gamma = 2^{G}$	the powerset of operations in the agent system

Collaborating Components

D–KnowledgeUpdate$_i$ enables the agent to update its current knowledge of the environment. D–KnowledgeUpdate takes the agent's current knowledge and the set of current stimuli and produces a set of focus and filter selectors. The focus and filter selectors are passed to the perception module that senses the environment to produce a new percept and update the agent's knowledge. D–KnowledgeUpdate is typed as follows:

$$D\text{–}KnowledgeUpdate_i : \mathcal{K}_i \times \mathcal{T}_i \rightarrow S_{\Theta_i} \times S_{\Phi_i}$$
$$D\text{–}KnowledgeUpdate_i(\kappa, \tau) = (s_\theta, s_\phi)$$

ActionSelection$_i$ selects operators. ActionSelection takes the current stimuli and the knowledge of the agent to select an operator and update the set of current stimuli. ActionSelection is typed as follows:

$$ActionSelection_i : \mathcal{T}_i \times \mathcal{K}_i \rightarrow Op_i \times \mathcal{T}_i$$
$$ActionSelection(\tau, \kappa) = (op, \tau')$$

ActionSelection encapsulates a behavior-based action selection mechanism extended with the notion of role and situated commitment. For the specification of a role and a situated commitment, see chapter 4, section 4.5.2.

Execution$_i$ converts the selected operator into an influence. Execution takes the operator and the current knowledge of the agent and produces an influence that is invoked in the environment. Execution is typed as follows:

$$Execution_i : Op_i \times \mathcal{K}_i \rightarrow Inf$$
$$Execution_i(op, \kappa) = inf$$

The *Collector* collects influences invoked by the agents, and adds the influences to the set of pending influences in the agent system. Collector is typed as follows:

$$Collector : \mathcal{I} \times Inf \rightarrow \mathcal{I}$$
$$Collector(\iota, inf) = \iota'$$

Collector adds influence *inf* to the set of pending influences, i.e. $\iota' = \iota \cup \{inf\}$.

OperationRestriction takes an influence, the current state of the application environment, and generates according to the set of action laws, an operation for the selected influence. OperationRestriction is typed as follows:

$$OperationRestriction : \mathcal{I} \times \Sigma \rightarrow \mathcal{I} \times G$$
$$OperationRestriction(\iota, \sigma) = (\iota', g')$$

OperationRestriction applies three functions in turn: (1) it selects an influence from the set of pending influences invoked by agents, (2) it converts the selected influence into an operation, and (3) it applies the actions laws to this operation.

The selection of an influence is defined by the *InfluenceSelection* function that is typed as follows:

$$InfluenceSelection : \mathcal{I} \times \Sigma \ \rightarrow \ Inf \times \mathcal{I}$$
$$InfluenceSelection(\iota, \sigma) = \ (inf, \iota')$$

The selection of an influence is based on the influence selection policy Pol_I that specifies the ordering of influences, taking into account the current state of the environment. The influence selection policy is not further specified, see variation mechanism CC4 (section 4.5.2.3). After influence selection holds: $\iota' = \iota \setminus \{inf\}$.

OperationGeneration converts the selected influence into an operation. OperationGeneration is typed as follows:

$$OperationGeneration : Inf \times \Sigma \rightarrow G$$
$$OperationGeneration(inf, \sigma) = g$$

The application of the perception laws is defined by the $ApplyL_A$ function that is typed as follows:

$$ApplyL_A : G \times \Sigma \rightarrow G$$
$$ApplyL_A(g, \sigma) = \ g'$$

$ApplyL_A$ applies the set of action laws L_A to operation g, given the current state of the application environment σ. $ApplyL_A$ restricts the operation g according to the set of applicable action laws. For the restricted g' holds:

$$\begin{array}{l} \forall s \in S_E : \\ \quad g'(s) = true \text{ iff} \\ \quad\quad (g(s) = true) \ \wedge (\forall\ la \in L_A : (\forall\ co \in la(g, \sigma) : co(s) = false)) \\ \quad g'(s) = false \text{ iff} \\ \quad\quad (g(s) = false) \ \wedge (\forall\ la \in L_A : (\forall\ co \in la(g, \sigma) : co(s) = false)) \\ \quad g'(s) = unspec \text{ otherwise} \end{array}$$

That is, (1) a state element of the environment is part of the target state of the restricted operation if the state element is part of the target state of the operation and none of the constraints of the applied action laws is applicable to the state element; (2) the restricted operation is invariable to the rest of the state elements of the environment.

For the restricted operation holds:

$$dom(g')^{\rightarrow true} = (dom(g)^{\rightarrow true}) \bigcap (\bigcup_{la \in L_A,\, co \in la(g,\sigma)} dom(co)^{\rightarrow false})$$
$$dom(g')^{\rightarrow false} = (dom(g)^{\rightarrow false}) \bigcap (\bigcup_{la \in L_A,\, co \in la(g,\sigma)} dom(co)^{\rightarrow false})$$
$$dom(g')^{\rightarrow unspec} = (dom(g)^{\rightarrow unspec}) \bigcup (\bigcup_{la \in L_A,\, co \in la(g,\sigma)} dom(co)^{\rightarrow true})$$

The target domain of the restricted operation (i.e. the subdomains of elements of state of the environment that map to true or false) consists of the intersection of the target domain of the original operation and the subdomain of elements of the state of the environment that are not constrained by the action laws.

The *Reactor* executes the restricted operation to the state of the application environment, and produces an operation to act in the deployment context if applicable. Reactor is typed as follows:

$$Reactor : G \times \Sigma \rightarrow \ \Sigma \times G$$
$$Reactor(g', \sigma) = (\sigma', g'')$$

The execution of the operation modifies the state of the application environment and produces an operation connected to the state of the deployment context that is forwarded to the translation module.

A.3.3 C & C Collaborating Components View Packet 3: Communication and Communication Service

A.3.3.1 Architectural Elements and Relationships

Fig. A.8 shows the collaborating components that realize the functionality for communication.

A.3.3.2 Specification of Architectural Elements

General Definitions

$\mathcal{M} = 2^M$	the powerset of messages in the agent system; $\mu \in \mathcal{M}$ denotes a set of messages
$\mathcal{D}dec$	the set of decoded message data, a decoded message data $md \in \mathcal{D}dec$ represents the data of a received message and is defined as a 4-tuple $\langle\, cid, sender, perform, content \,\rangle$; the elements of this tuple are defined by the communication language L (section A.1.2); we use the notation $tuple\|_{element}$ to selects an element of a tuple
$\mathcal{D}enc$	the set of message data to encode messages, a decoded message data $md \in \mathcal{D}dec$ is defined as $\langle\, cid, addressees, perform, content \,\rangle$; the elements of this tuple are defined by the communication language L
D	the set of conversations in the agent system; a conversation $d \in D$ is defined as a 3-tuple $\langle\, pname, cid, history \,\rangle$ with $pname$ the name of the protocol of the conversation, $cid \in Cid$ the unique id of the conversation, and $history$ the sequence of message data of received and sent messages of the conversation

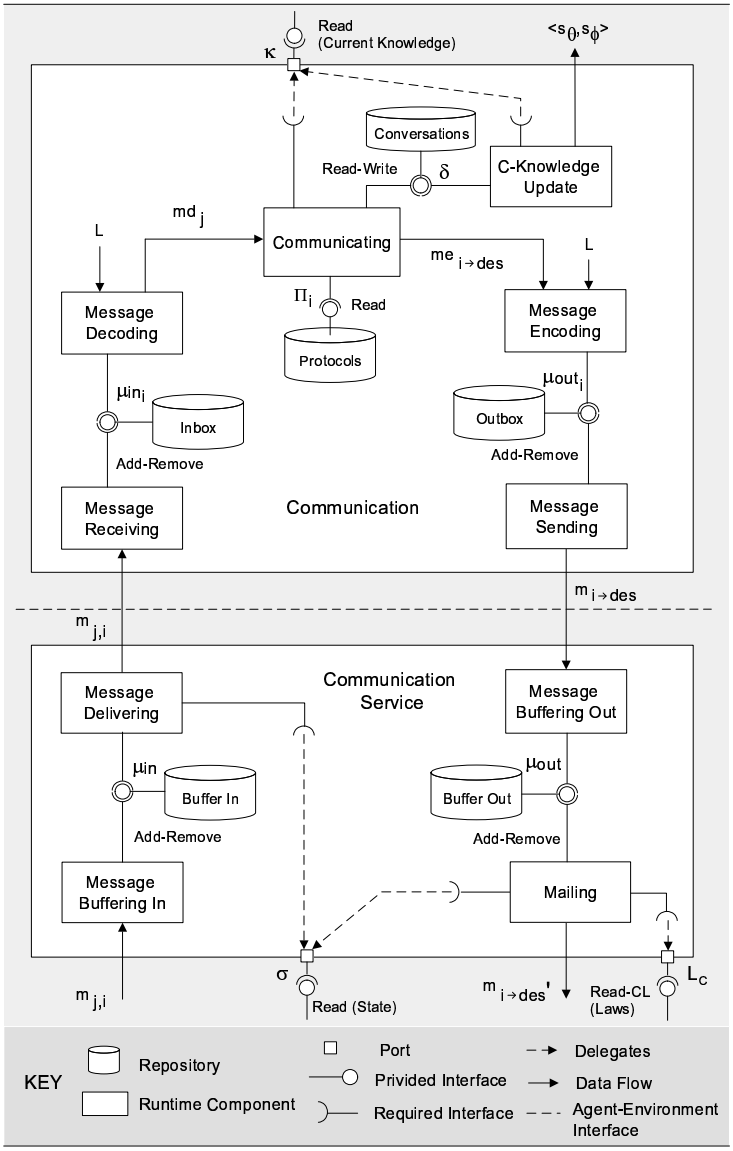


Figure A.8: Collaborating components of communication & communication service

$D_i \subseteq D$	the set of all conversations of agent a_i
$\Delta_i \subset 2^{D_i}$	the powerset of conversations of agent a_i

St	the set of protocol steps in the agent system; a protocol step $st \in St$ is defined as a tuple of functions $\langle\, cond, step \,\rangle$; $cond$ is a boolean function that determines whether the protocol step is applicable, and $step$ determines the effects of the protocol step (see below)
$pr \in \Pi_i$	a communication protocol of agent a_i that is defined as a tuple $pr = \langle\, pname, \langle\, st_1, \ldots, st_p \,\rangle \,\rangle$ with $pname$ the protocol name and $st_j \in St$ for all $j \in \{1..p\}$; for the application-specific definition of protocols, see variation mechanism CC7 (section 4.5.3.3)

Collaborating Components

First we elaborate on the communicating component, after that we specify the other components in the collaboration.

$Communicating_i$ handles the agent's communicative interactions. Communicating processes incoming messages and produces outgoing messages according to well-defined communication protocols. Communicative interactions can modify the agent's state possibly affecting the agent's selection of influences; an important example is the activation and deactivation of a situated commitment. A communication protocol $pr \in \Pi_i$ consists of a series of protocol steps (see specification above). We distinguish between three types of protocol steps: (A) conversation initiation, (B) conversation continuation, and (C) conversation termination. We specify each type of protocol step in turn.

A. Conversation initiation starts a new conversation according to a particular protocol. A conversation can be initiated by the agent itself ($EndogenousInitiation$) or by another agent ($ExogenousInitiation$).

$$EndogenousInitiation = \langle\, EndoIniCond, EndoIniStep \,\rangle$$

Applied to an agent with identity $i \in Y$:

$$EndoIniCond_i : \mathcal{K}_i \rightarrow \ Bool$$
$$EndoIniStep_i : \Delta_i \times \mathcal{K}_i \rightarrow \ \mathcal{D}enc \times \Delta_i \times \mathcal{K}_i$$

$EndoIniCond_i$ is the $cond$ function of the protocol step and $EndoIniStep_i$ the $step$ function. If $EndoIniCond_i(\kappa) = true$ then the protocol step $EndoIniStep_i(\delta, \kappa) = (me, \delta', \kappa')$ is applicable. $EndoIniStep_i$ produces the message data me for a new message, it updates the agent's current knowledge, and it adds a new conversation to the set of ongoing conversations. For the resulting set of conversations holds:

$$\delta' = \delta \cup \{d\} \text{ with } d = \ \langle\, pr, Cid(), \langle\, me \rangle \rangle$$

That is, a new conversation d is added to the set of ongoing conversations of the agent. $EndoIniStep_i$ updates the agent's current knowledge, i.e. κ' includes

modifications of the agent state implied by $EndoIniStep_i$.

ExogenousInitiation is defined as follows:

$$ExogenousInitiation = \langle\, ExoIniCond, ExoIniStep \,\rangle$$

Applied to an agent with identity $i \in Y$:

$$ExoIniCond_i : \mathcal{D}dec \times \mathcal{K}_i \rightarrow Bool$$
$$ExoIniStep_i : \mathcal{D}dec \times \Delta_i \times \mathcal{K}_i \rightarrow \Delta_i \times \mathcal{K}_i$$

$ExoIniCond_i$ is the *cond* function of the protocol step and $ExoIniStep_i$ the *step* function. If $ExoIniCond_i(md, \kappa) = true$ then $ExoStep_i(md, \delta, \kappa) = (\delta', \kappa')$ is applicable. $ExoIniStep_i$ updates the agent's current knowledge and adds a new conversation to the set of ongoing conversations. For the updated set of conversations holds:

$$\delta' = \delta \cup \{d\} \text{ with } d = \langle\, md\,|_{pr}, md\,|_{cid}, \langle\, md \,\rangle\rangle$$

That is, a new conversation d is added to the set of ongoing conversations of the agent, with $md\,|_{pr} \in \Pi_i$ the protocol of the conversation, $md\,|_{cid}$ the conversation id, and $\langle\, md \,\rangle$ the initial history of the new conversation. $ExoIniStep_i$ updates the agent current knowledge, i.e. κ' includes modifications of the agent state implied by $EndoIniStep_i$.

B. Conversation continuation performs a step in an ongoing conversation. We distinguish between three types of conversation continuations: *Process* deals with a received message data without directly reacting to it, *Reply* immediately reacts to a received message data, and finally *Continue* picks up the conversation after a break. We specify each type in turn. *Process* is defined as follows:

$$Process = \langle\, ProcessCond, ProcessStep \,\rangle$$

Applied to an agent with identity $i \in Y$:

$$ProcessCond_i : \mathcal{D}dec \times \Delta_i \times \mathcal{K}_i \rightarrow Bool$$
$$ProcessStep_i : \mathcal{D}dec \times \Delta_i \times \mathcal{K}_i \rightarrow \Delta_i \times \mathcal{K}_i$$

$ProcessCond_i$ is the *cond* function of the protocol step and $ProcessStep_i$ the *step* function. If $ProcessCond_i(md, \delta, \kappa)$ returns *true* then $ProcessStep_i(md, \delta, \kappa) = (\delta', \kappa')$ is applicable. For the updated set of conversations holds:

$$\delta' = (\delta \backslash \{d\}) \cup \{d'\}$$
$$\text{with } d \in \delta : md|_{cid} = d|_{cid} \text{ and } d' = \langle\, d|_{pr}, d|_{cid}, \langle\, d|_{hist} \circ md \,\rangle\rangle$$

Process adds the message data md to the history of the conversation to which the processed message belongs. Furthermore, *Process* updates the agent current

knowledge, i.e. κ' includes modifications of the agent's state implied by the protocol step.

Reply is defined as follows:

$$Reply = \langle\, ReplyCond, ReplyStep \,\rangle$$

Applied to an agent with identity $i \in Y$:

$$ReplyCond_i : \mathcal{D}dec \times \Delta_i \times \mathcal{K}_i \rightarrow \ Bool$$

$$ReplyStep_i : \mathcal{D}dec \times \Delta_i \times \mathcal{K}_i \rightarrow \ \mathcal{D}enc \times \Delta_i \times \mathcal{K}_i$$

$ReplyCond_i$ is the *cond* function of the protocol step and $ReplyStep_i$ the *step* function. If $ReplyCond_i(md, \delta, \kappa) = true$ then $ReplyStep_i(md, \delta, \kappa) = \ (me, \delta', \kappa')$ is applicable. me is the message data of the agents' reply to the received message with message data md. For the updated set of conversations holds:

$$\delta' = (\delta \backslash \{d\}) \cup \{d'\}$$
$$\text{with } d \in \delta : md|_{cid} = d|_{cid} \text{ and } d' = \langle\, d|_{pr}, d|_{cid}, \langle\, (d|_{hist} \circ md) \circ me \,\rangle \rangle$$

Reply adds the received message data md and the composed message data me to the history of the conversation to which the received message belongs (i.e. $d|_{hist}$). Furthermore, *Reply* updates the agent current knowledge according to the modifications implied by the *Reply* step.

Continue is defined as follows:

$$Continue = \langle\, ContCond, ContStep \,\rangle$$

Applied to an agent with identity $i \in Y$:

$$ContCond_i : D_i \times \mathcal{K}_i \rightarrow \ Bool$$

$$ContStep_i : D_i \times \mathcal{K}_i \rightarrow \ \mathcal{D}enc \times D_i \times \mathcal{K}_i$$

$ContCond_i$ is the *cond* function of the protocol step and $ContStep_i$ is the *step* function of *Continue*. When $ContCond_i(d, \kappa)$ becomes *true* the agent continues conversation d with $ContStep_i(d, \kappa) = (me, d', \kappa')$. me is the message data to compose the new message and κ' is the agent's updated current knowledge implied by the protocol step. For the updated conversation holds:

$$d' = \langle\, d|_{pr}, d|_{cid}, \langle\, d|_{hist} \circ me \,\rangle \rangle$$

Continue adds the composed message data me to the history of the conversation for which the protocol step is applicable. After the *Continue* step, the set of ongoing conversations of the agent is updated to $\delta' = (\delta \backslash \{d\}) \cup \{d'\}$.

C. Conversation termination concludes an ongoing conversation. The termination of a conversation can be induced by changing circumstances in the environment or it can directly follow a preceding step of the conversation such as a *Reply* or a *Continue*. *Terminate* is defined as follows:

$$Terminate = \langle\, TermCond, TermStep \,\rangle$$

Applied to an agent with identity $i \in Y$:

$$TermeCond_i : D_i \times \mathcal{K}_i \rightarrow Bool$$
$$TermStep_i : D_i \times \mathcal{K}_i \rightarrow \mathcal{K}_i$$

$TermCond_i$ is the *cond* function and $TermStep_i$ is the *step* function of the $Terminate$ protocol step. If $TermCond_i(d, \kappa)$ returns *true* then $TermStep_i(d, \kappa) = \kappa'$ terminates the conversation d and updates the agent's current state accordingly. After the $Terminate$ step, the set of ongoing conversations of the agent is updated to $\delta' = \delta \backslash \{d\}$.

For the application-specific definition of communication protocols, see variation mechanism CC7 (section 4.5.3.3). Now we look at the functionalities of the other collaborating components of communication and communication service.

$C\text{–}KnowledgeUpdate_i$ enables the agent to update its current knowledge according to its ongoing conversations. C–KnowledgeUpdate takes the agent current knowledge and the set of conversations and produces a set of focus and filter selectors that are passed to the perception module that senses the environment to produce a new percept and update the agent's knowledge. C–KnowledgeUpdate is typed as follows:

$$C\text{–}KnowledgeUpdate_i : \mathcal{K}_i \times \Delta_i \rightarrow S_{\Theta_i} \times S_{\Phi_i}$$
$$C\text{–}KnowledgeUpdate_i(\kappa, \delta) = (s_\theta, s_\phi)$$

$MessageEncoding_i$ encodes newly composed message data into messages and puts the messages in the outbox buffer of the agent. Message encoding is based on the communication language L that is shared among the agents in the system. MessageEncoding is typed as follows:

$$MessageEncoding_i : (\mathcal{D}enc \rightarrow M) \times \mathcal{M} \rightarrow \mathcal{M}$$
$$MessageEncoding_i(Encode(me), \mu out_i) = \mu out_i'$$

After message encoding, the new message is added to the outbox, i.e.:

$$\mu out_i' = \mu out_i \cup \{m\} \text{ with } m = Encode(me)$$

$MessageSending_i$ selects a message from the set of pending messages in the outbox buffer and passes it to the communication service. Message selection is typically first-in-first-out, see variation mechanism CC8 (section 4.5.3.3). MessageSending is typed as follows:

$$MessageSending_i : \mathcal{M} \rightarrow M \times \mathcal{M}$$
$$MessageSending_i(\mu out_i) = (m_{i \rightarrow des}, \mu out_i')$$

Message sending removes the sent message from the outbox, i.e.:

$$\mu out_i' = \mu out_i \backslash \{m_{i \to des}\}$$

MessageBufferingOut collects messages sent by agents and puts them in the output buffer of the communication service. MessageBufferingOut is typed as follows:

$$MessageBufferingOut : M \times \mathcal{M} \to \mathcal{M}$$
$$MessageBufferingOut(m_{i \to des}, \mu out) = \mu out'$$

For the outbox holds $\mu out' = \mu out \cup \{m_{i \to des}\}$.

Mailing transmits a message selected from the output buffer, taking into account the current state of the application environment and the set of communication laws. Mailing is typed as follows:

$$Mailing : \mathcal{M} \times \Sigma \to \mathcal{M} \times \Sigma$$
$$Mailing(\mu out, \sigma) = (m_{i \to des'}, \mu out', \sigma')$$

Mailing applies three functions in turn: (1) it selects a message from the set of pending messages sent by agents, (2) it applies the communication laws to the selected message, and (3) it transmits the message in the deployment context if applicable.

The selection of an influence is defined by the *MessageSelection* function that is typed as follows:

$$MessageSelection : \mathcal{M} \times \Sigma \to M \times \mathcal{M}$$
$$MessageSelection(\mu out, \sigma) = (m_{i \to des}, \mu out')$$

The selection of a message from the output buffer is based on the message selection policy Pol_C that specifies the ordering of messages, taking into account the current state of the environment. The message selection policy is not further specified, see variation mechanism CC8 (section 4.5.3.3). After message selection holds: $\mu out' = \mu out \setminus \{m_{i \to des}\}$.

The application of the communication laws is defined by the $ApplyL_C$ function that is typed as follows:

$$ApplyL_C : M \times \Sigma \to M$$
$$ApplyL_C(m_{i \to des}, \sigma) = m_{i \to des'}$$

$ApplyL_C$ applies the set of communication laws L_C to message $m_{i \to des}$, given the current state of the application environment σ. For the resulting message $m_{i \to des'}$ holds:

$$\forall\, y \in des' : \\ (y \in des) \ \wedge \ (\forall\, lc \in L_C : (\forall\, co \in lc(m_{i \to des}, \sigma) : co(y) = false))$$

That is, the message $m_{i \to des'}$ will be transmitted to all addressees of the original message that are not constrained by any of the communication laws.

MessageBufferingIn collects messages from the deployment context and puts them in the input buffer of the environment. MessageBufferingIn is typed as follows:

$$MessageBufferingIn : M \times \mathcal{M} \to \mathcal{M}$$
$$MessageBufferingIn(m_{j,i}, \mu in) = \mu in'$$

For the input buffer holds: $\mu in' = \mu in \cup \{m_{j,i}\}$.

MessageDelivering delivers the messages of the input buffer to the appropriate agents. MessageDelivering is typed as:

$$MessageDelivering : \mathcal{M} \times \Sigma \to \ M \times \mathcal{M}$$
$$MessageDelivering(\mu in, \sigma) = \ (m_{j,i}, \mu in')$$

Message $m_{j,i}$ originating from agent a_j is delivered to agent a_i, the addressee of the message. The message is removed from the input buffer, i.e. $\mu in' = \mu in \setminus \{m_{j,i}\}$.

MessageReceiving$_i$ accepts messages and puts them in the agent's inbox. MessageReceiving is typed as:

$$MessageReceiving_i : M \times \mathcal{M} \to \mathcal{M}$$
$$MessageReceiving_i(m_{j,i}, \mu in_i) = \mu in_i'$$

The received message $m_{j,i}$ is added to the inbox μin_i, i.e. $\mu in_i' = \mu in_i \cup \{m_{j,i}\}$.

Finally, *MessageDecoding*$_i$ selects a message from the agent's inbox and decodes the message according to the given communication language. MessageDecoding is typed as follows:

$$MessageDecoding_i((\mathcal{M} \to M) \times \Lambda \ \to \mathcal{M} \times \mathcal{D}dec$$
$$MessageDecoding_i(SelectMsg(\mu in), L) = \ (\mu in', md_j)$$

The function $SelectMsg(\mu in) = m_{j,i}$ selects a message from the agent's inbox. For the application-specific definition of $SelectMsg$, see variation mechanism CC8 (section 4.5.3.3). The decoded message data of the selected message is passed to the Communicating component that will process it. After message decoding the message is removed from the inbox, i.e. $\mu in' = \mu in \setminus \{m_{j,i}\}$.

Appendix B

A Framework for Situated Multiagent Systems

This appendix gives an overview of an object-oriented framework that supports the main functionalities of the reference architecture for situated multiagent systems described in chapter 4. The overview consists of four parts. Part B.1 presents the main packages of the framework and discusses the two main parts of the framework: agent and application environment. Part B.2 zooms in on decision making with a free-flow tree, and part B.3 explains how simultaneous actions are supported in the framework. Finally, part B.4 shows how the framework is applied to an experimental robot application. For a detailed documentation of the framework, we refer to [92, 229].

B.1 General Overview of the Framework

A framework consists of a core (also called frozen-spot) that is common to all applications derived from the framework, and hot-spots that represent the variable parts which allow a framework to be adapted to a particular application [37, 73]. Fig. B.1 shows a general overview of the packages of the framework for situated multiagent systems. The `Agent` and `Application Environment` packages encapsulate the core of the framework and provide factories to create agents and the application environment. The `Shared` package encapsulates helper classes for `Agent` and `Application Environment`.

Developing an application with a software environment starts with the implementation of the various hot spots of the `Agent` and `Application Environment` package (we discuss the hot spots below). `SystemCreator` then integrates the hot spots with the framework core to build the entire application. `SystemCreator` creates the application environment and populates it with the application agents.

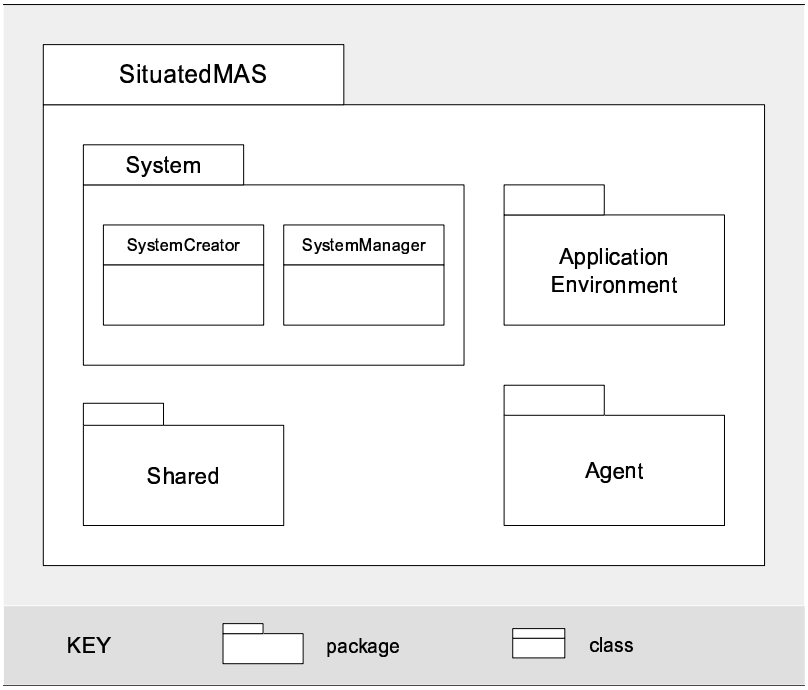


Figure B.1: General overview of the framework

`SystemCreator` returns an instance of `SystemManager` that is used to control the execution of the application. `SystemManager` allows the user to start the application, to suspend and resume the execution, and to terminate the application.

To develop an application with agents deployed in a physical environment, only the hot spots of the Agent package have to be implemented and integrated with the framework core (Agent package). The integrated software can directly be deployed on the physical machines. To enable the agents to interact with the physical environment, the software has to be connected to sensors and actuators.

B.1.1 Overview of the Agent Package

Fig. B.2 shows a general overview of the `Agent` package. The package is divided in several sub-packages, we briefly explain each of the sub-packages in turn.

`KnowledgeIntegration` encapsulates the agent's internal state that is modelled as a collection of knowledge objects (`KnowledgeObject`). Besides basic support for adding and removing knowledge objects, `KnowledgeIntegration` provides various additional features such as support to update the state with a given set of knowl-

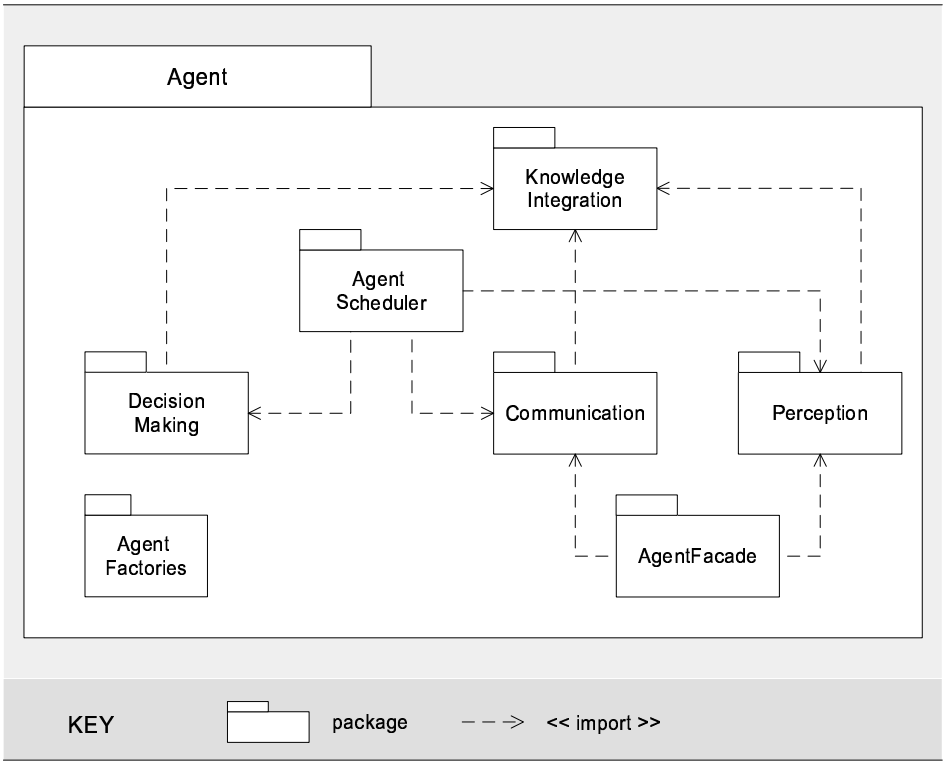


Figure B.2: General overview of the Agent package

edge objects, selection of the knowledge objects of a particular type, registration of an observer to notify changes of a selected type of knowledge objects, etc.

`Perception` enables the agent to sense the environment. `Perception` supports selective perception, i.e. agents can sense the environment with a set of selected foci (`Focus`), interpret representations with descriptions (`Description`), and filter the resulting `Percept` with a set of selected filters (`Filter`). `Perception` interacts with the environment via a set of sensors (`Sensor`). For agents situated in a software environment, a sensor is an abstraction that provides an interface with the application environment. Software agents receive the representation of a perception request via the `AgentFacade`. For agents situated in a physical environment, a sensor is the physical device the agent uses to sense the surrounding world.

`Communication` deals with the communicative interactions of the agent. Agents communicate according to well-defined protocols (`Protocol`). In addition to the

various protocol steps specified in the reference architecture, the framework provides support for time-outs. The developer can associate a time duration to a conversation, together with a reaction. When no activity has occurred in the conversation for the specified time duration, the accompanying reaction will be executed. For example, if an agent does not receive an answer to a request within a particular time window it can repeat the request, it can discard the conversation, or it can react in some other way. Messages are exchanged in `ExternalMessage` format. An `ExternalMessage` encapsulates a message as a plain string. Internally agents use `AgentMessage` to represent a message. Instances of `AgentMessage` encode messages in terms of domain objects. The conversion of messages is handled by the `SLDecoderEncoder` and is based on a domain `Ontology`. "SL" stands for Semantic Language and is defined by FIPA [77]. For the implementation of the `SLDecoderEncoder`, we reused a package of the Jade libraries [38]. `Ontology` represents the common vocabulary agents use to communicate with one another. Each concept that agents want to use as content of a message needs to be included in their ontology. A concept is stored in the vocabulary as a tuple of the class of the concept and an external name used to refer to the concept (`Tuple<String, Class>`). For example, for agents in the Packet-World, the ontology is defined as:

```
Ontology ont = new Ontology();
ont.addToVocabularium(PacketRepresentation.class, "packet");
ont.addToVocabularium(PositionRepresentation.class, "position");
...
ont.addToVocabularium(Head.class, "head");
...
```

`Communication` is equipped with a `Transceiver` to exchange messages with other agents. For agents situated in a software environment, the transceiver is an abstraction that provides an interface with the application environment. Software agents receive incoming messages via the `AgentFacade`. For agents situated in a physical environment, the transceiver is the physical device the agent uses to communicate with other agents in their neighborhood.

`DecisionMaking` encapsulates a behavior-based action selection mechanism that supports the notion of role (`Role`) and situated commitment (`SituatedCommitment`). The framework offers the application developer the `FreeFlow` package to instantiate free-flow trees. We elaborate on decision making with a free-flow tree in section B.2. Decision making results in the selection of an `Operator` that is passed to `Execution`. For agents situated in a software environment, the execution module is an abstraction that converts the operator into an `Influence` that is invoked in the application environment. For agents situated in a physical environment, the execution module interfaces with the physical device the agent uses to act in the

environment, such as a switch or a motor.

`AgentScheduler` encapsulates the thread of the agent. `AgentScheduler` determines when the different modules (perception, decision making, and communication) get control. Decision making and communication can select foci and filters that are used by perception to perceive the environment when it gets control. Scheduling of the various activities can be customized according to the requirements at hand. The framework offers a default schema `LTDSchedule` (Look—Talk—Do [202]) that extends `AgentSchedule`. `LTDSchedule` successively activates perception, communication, decision making in an endless loop.

`AgentFactories` is a package that supports the creation of agents. `AgentFactories` consists of two sub-packages: `SoftwareAgentFactory` and `PhysicalAgentFactory` that can be used to create agents situated in a software environment and a physical environment respectively. In particular, the `PhysicalAgentFactory` supports the instantiation of robot software with the Lego-Mindstorms package [10].

Hot Spots. The hot spots of `Agent` can be divided in two groups: hot spots related to the interaction of the agent with the environment, and hot spots related to the agent's behavior.

Hot spots related to the interaction with the environment are only applicable for agents situated in a physical environment and include: `Sensor`, `Transceiver`, and `Execution`. For a concrete application, these hot spots are instantiated as a means to interface with the appropriate physical devices. For agents that live in a software environment the core of the framework encapsulates general implementations for sensor, transceiver and execution that are used for the interfacing with the application environment. We illustrate hot spots related to the interaction with the environment for a robot in section B.4.

Hot spots related to the behavior of the agent determine how an agent perceives the environment, how it selects actions, and how it communicates with other agents. The hot spots include: `KnowledeObject`, `Focus`, `Description`, `Percept`, `Filter`, `DecisionMaking`, `Role`, `SituatedCommitment`, `Operator`, `Ontology`, `Protocol`, and `AgentSchedule`. We illustrate a number of instances of these hot spots in section B.2.

B.1.2 Overview of the Application Environment Package

Fig. B.3 shows a general overview of the application environment package. We briefly look at the various sub-packages.

`EnvironmentFacade` shields the internals of the application environment to agents. The facade provides an interface to agents to sense the application environment,

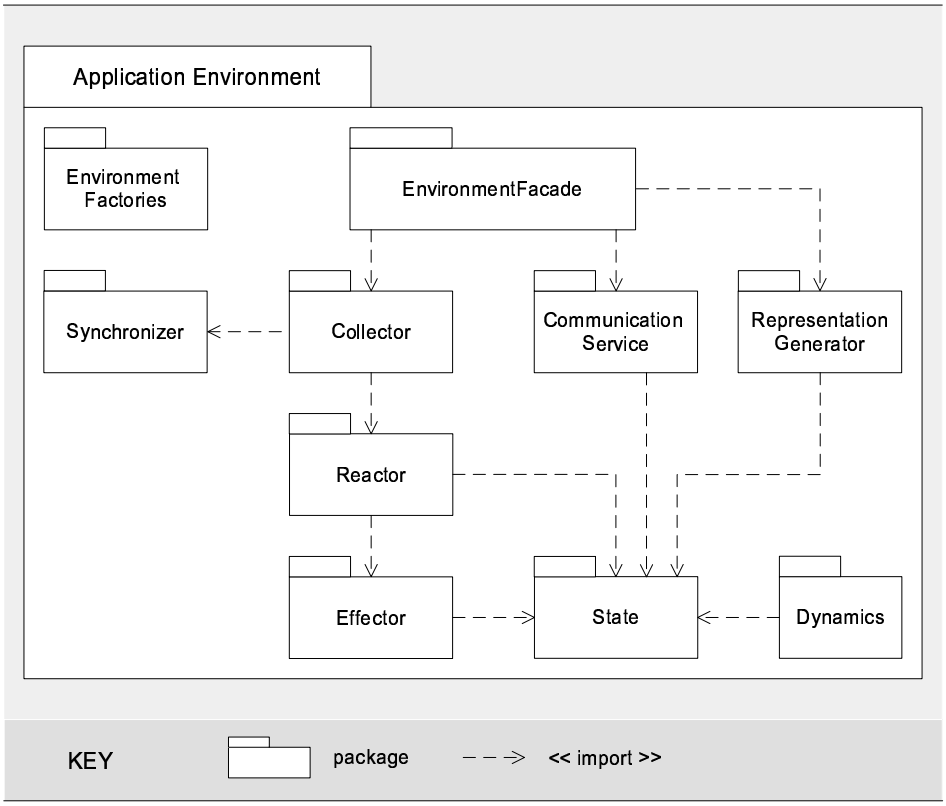


Figure B.3: General overview of the Application Environment package

to invoke influences, and to send messages. `EnvironmentFacade` dispatches the various activities for processing to the appropriate modules.

`State` encapsulates the actual state of the application environment. The state of the application environment includes a representation of the topology of the environment, state of static and dynamic objects, external state of agents (e.g., identities and positions), and state of environmental properties that represent system-wide characteristics. An example of an environmental property in the Packet-World is a gradient field that guides agents to a battery charger. State in the framework is set up as a collection of `Item` objects and a collection of `Relation` objects. `Item` is an abstraction for elements in the application environment, with `StaticItem` and `DynamicItem` as specializations. The state of a `StaticItem` is invariable over time, state of a `DynamicItem` may change over time. `Relation` represents a relation between two or more `Item` objects. An example of a `Relation`

in the Packet-World is an agent that holds a packet. The framework supports various methods to observe and manipulate `Item` and `Relation` objects.

`Dynamics` encapsulates a collection of ongoing activities (`OngoingActity`). An ongoing activity defines the behavior of a `DynamicItem` taking into account the current state of the application environment. Ongoing activities can be created at system startup and exist for the full lifetime of the application, or they can be created and destroyed dynamically at runtime. An example of an ongoing activity in the Packet-World is the maintenance process of digital pheromones. `OngoingActity` is supplied with an `OngoingActivitySchedule` that encapsulates a thread to execute the ongoing activity. Developers can extend `OngoingActivitySchedule` to define application-specific behavior for ongoing activities.

`RepresentationGenerator` is an active module that generates representations (`Representation`) of the state of the application environment for agents. Representations are generated according to representation requests and the applicable perception laws that govern what the agents can observe. A concrete perception law is defined as a subclass of `PerceptionLaw` and must implement the method:

```
public abstract Representation enforce(AgentId observer,
Representation representation, Vector<Focus> foci, State state);
```

A concrete perception law puts application specific restrictions on the representation generated for an observer, given the set of selected foci and the current state of the application environment. The following example illustrates a perceptual law for the Packet-World:

```
public Representation enforce(AgentId observer,
 Representation representation, Vector<Focus> foci, State state) {
  Representation repr = copy(representation);
  Position observerpos = ((GridState)state).getPosition(observer);
  if (containsVisualFocus(foci)) then
    for (int i=0; i<((GridStateRepresentation)repr).nbItems(); i++){
     ItemRepresentation item = repr.getItem(i);
     Position itempos = item.getPosition(i);
     if (item.isVisible()
      && ((GridState)state).obstacleBetween(observerpos, itempos))
     then removeItem(representation, item);
    }
  return representation;
}
```

This law removes all the visible items in a representation that are out of the view

of an observer due to an obstacle.

`RepresentationGenerator` applies the perception laws in a strict sequence. The definition of the ordering is a responsibility of the developer.

`CommunicationService` is an active module that handles message transport through the environment. Messages are delivered first-in-first-out. The application developer can define communication laws that enforce domain specific constraints on the transport of messages. A concrete communication law is defined as a subclass of `CommunicationLaw` and must implement the method:

```
public abstract ExternalMessage enforce(AgentId sender,
   ExternalMessage message, State state);
```

A typical example is a communication law that restricts the delivering of messages to a specific distance from the sender:

```
public ExternalMessage enforce(AgentId sender,
 ExternalMessage message, State state) {
  ExternalMessage msg = copy(message);
  for (int i=0; i<msg.nbAddressees(); i++){
   if (!((GridState)state).withinComRange(sender,msg.addressee(i)))
   then removeAddressee(message,msg.addressee(i));
  return message;
}
```

This laws drops all addressees of the message that are not within communication range of the sender.

`Synchronizer` determines the type of synchronization of simultaneous actions. Simultaneous actions are actions that happen together and that can have a combined effect in the application environment. An example of simultaneous actions in the Packet–World are two agents that push the same packet in different directions. As a result, the packet moves according to the resultant of the two actions. Ferber and Müller have introduced a model for simultaneous actions in which all the agents of the multiagent system act at one global pace, i.e. the agents are globally synchronized [76]. In this model, the environment combines the influences of all agents in each cycle to deduce a new state of the environment. In [203, 207, 205], we have introduced a model for regional synchronization. With regional synchronization agents form synchronized groups—regions—on the basis of their actual locality. Different regions can act asynchronously, while agents act synchronously within their region. Regional synchronization complies with the basic characteristic of locality of situated agents. We elaborate on support for simultaneous action in section B.3.

The framework includes support for three types of synchronization: (1) `NoSynchronization`, i.e. agents act asynchronously which implies that there is no support for simultaneous actions; (2) `GlobalSynchronization`, all agents act at a global pace—i.e. the Ferber–Müller model; and (3) `RegionalSynchronization`, i.e. agents act simultaneously based on their actual locality.

`Collector` is an active module that collects the influences (`Influence`) invoked by agents. `Collector` uses a `Synchronizer` to determine the sets of synchronized agents. If no synchronization is provided, the collector directly passes the influences to the reactor. With global synchronization, the collector collects the influences of all agents in the system before it passes the complete set to the reactor. With regional synchronization, the collector passes sets of influences per region to the reactor. `Collector` encapsulates its own thread, so that it can execute influences in parallel with other activities in the application environment.

`Reactor` and `Effector`. `Reactor` is responsible for processing sets of synchronized influences. The reactor *calculates* the effects of the influences according to current state of the application environment and the action laws of the multiagent system. This results in a set of effects (`Effect`). `Effector` is responsible for *executing* the effects of the influences resulting in state changes in the application environment. We elaborate on interaction in the application environment with the `Collector-Reactor-Effector` chain in section B.3.

`EnvironmentFactory` is a package that supports developers with the creation of an application environment for the multiagent system. `EnvironmentFactory` creates the internals of the application environment, it initializes the state of the environment with items and relations between items, it integrates the laws for perception, action, and communication, and it allows the developer to specify a synchronization approach for the application.

Hot Spots. The hot spots of the application environment include: `State` with `StaticItem`, `DynamicItem` and `Relation`, `OngoingActivity`, `Representation`, `Influence`, and `Effect`. Besides, `PereceptionLaw`, `ActionLaw`, and `CommunicationLaw` are hot spots that have to be defined for the application at hand. Finally, `Synchronizer` is a hot spot of the application environment for which the developer can simply select one of the available synchronizers.

B.2 Decision Making with a Free-Flow Tree

In this section, we explain in detail how free-flow trees extended with roles and situated commitments are supported by the framework. For an in dept discussion of free-flow trees we refer to chapter 3, section 3.4.3.2.

Fig. B.4 shows the main classes of the `FreeFlow` package of the framework.

A free-flow tree consists of three types of tree elements (`TreeElement`): `Node`,

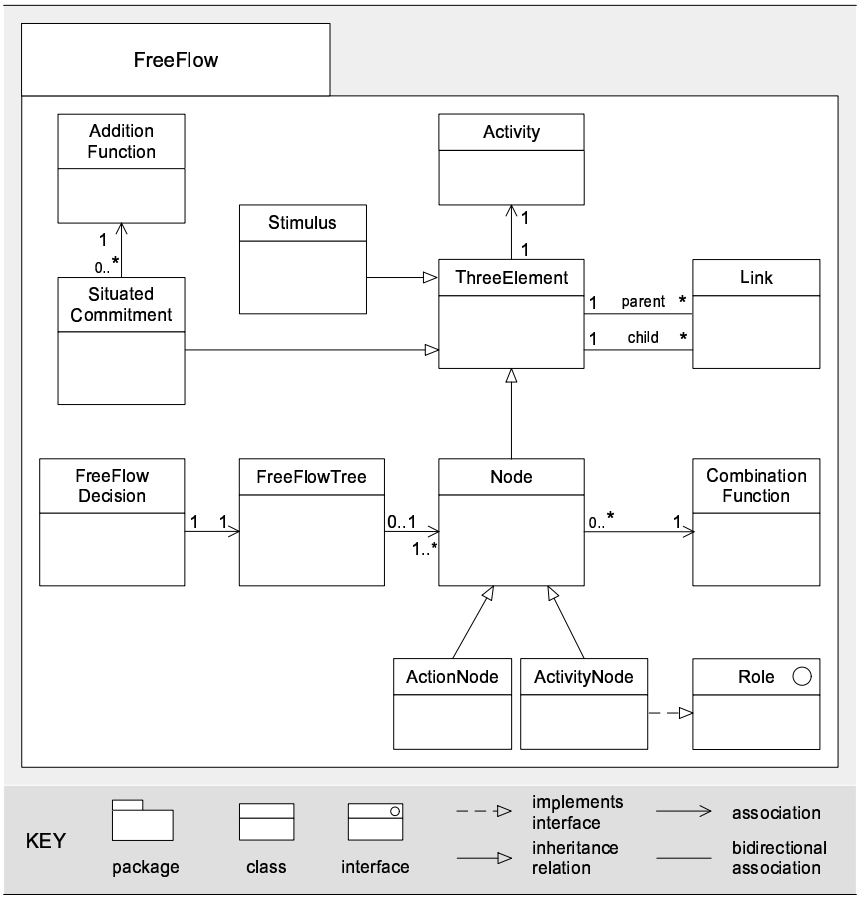


Figure B.4: Overview of the Free-flow package

`Stimulus`, and `SituatedCommitment`. Tree elements are connected through links (`Link`). A tree element receives an amount of activity (represented by `Activity`) of its parent elements and can inject activity in its child elements. Each link has a weight factor that determines how much of the injected activity is passed along that link. `Stimulus` has no parent elements but calculates its activity based on the internal state of the agent. `SituatedCommitment` is connected with a non-empty set of nodes that represents its source roles, and one particular node that represents its goal role. A situated commitment can be triggered by two conditions: one that activates the commitment and one that deactivates it. In the activated state, the commitment combines the activity receive from its source roles with an

`AdditionFunction` and injects the resulting activity in its goal role. `Node` is further specialized in `ActivityNode` and `ActionNode`. An `ActivityNode` is a regular node of the tree, an `ActionNode` is a leave node of the tree that is associated with an operator. Each `Node` has a `CombinationFunction` that determines how the activity received from its parent nodes is combined. Activity nodes that represent top nodes of a role have to implement the `Role` interface that associates an explicit name with the role. A `FreeFlowTree` represents the system node and is the entry point for `FreeFlowDecision` that implements the action selection algorithm.

Hot Spots. To build a concrete free-flow tree a number of hot spots have to be implemented. In particular, `Activity`, `Stimulus`, `SituatedCommitment`, `AdditionFunction` and `CombinationFunction`, `ActionNode`, `Role`, and `Link` are hot spots. The framework supports the developer with various basic implementations for most of these hot spots. `BasicActivity` is a subclass of `Activity` that represents a basic representation of activity by means of a double value. More advanced implementations have to be defined by the developer. The framework supports the definition of simple stimuli (`SimpleStimulus`) as well as multi-directional stimuli (`VectorStimuli`). A situated commitment has to be defined as a subclass of `SituatedCommitment` and requires the definition of an activation condition, a deactivation condition, and the definition of the outcome of the situated commitment when it is activated. This latter requires the definition of a concrete `AdditionFunction`. As an example, we illustrate the definition of the situated commitment Charging of a Packet-World agent:

```
public class Charging extends SituatedCommitment {
 private int toCharge;
 private int charged;
 public Charging(ActivityNode goalRole, Vector<ActivityNode>
   sourceRoles, int toCharge, int charged) {
    super("charging", goalRole, sourceRoles);
    this.toCharge = toCharge;
    this.charged = charged;
 }
 public boolean activationCondition(KnowledgeIntegration knowledge) {
    if (knowledge.getEnergyLevel() < toCharge)
       return true;
    else
       return false;
 }
 ...
 public Activity calculateActivity(KnowledgeIntegration knowledge) {
    if (isActivated()) {
       PositiveActivitiesAddtionFunction addFunction =
```

```
            new PositiveActivitiesAdditionFunction(sourceRoles.size()));
         return addFunction.calculateActivation(this);
      }
      else
         return new BasicActivity(0);
   }
   ...
}
```

For each `Node` a `CombinationFunction` has to be defined. The framework supports various basic functions, including `AddFunction`, `MultiplyFunction`, and `MaximumFunction`. `ActionNode` is a hot spot, for each action node of the tree a subclass of `ActionNode` has to be defined. Such subclass must implement the method `getOperator()`. This method returns the operator that is executed by the agent if that particular node is selected for execution. Each `Link` in the tree has to be assigned a weight factor, except if the weight factor has the default value of 1. Finally, a subclass of `FreeFlowDecision` must be defined. This subclass must implement the abstract method `createFreeFlowTree()` that constructs the application specific free-flow tree.

The framework's cookbook [229] provides examples of the various hot spots to support the developer with the instantiation of a free-flow tree. Still, designing a free-flow tree for a non-trivial agent remains a complex matter. In [185], we have described a modeling language and design process to support the design of free-flow trees with roles and situated commitments. At the highest level, roles and their interdependencies are caught into high level role models. These role models are used as a basis for designing a skeleton of the free-flow architecture. Next, statechart-based models are defined for the basic roles [95]. These statecharts allow to refine the skeleton tree. The resulting free-flow tree design provides a detailed description to instantiate the various hot spots of the framework. Yet, determining the various combination functions and the values of the weight–factors of links remain a tricky job.

B.3 Simultaneous Actions in the Environment

An interesting feature provided by the framework is support for simultaneous actions. Support for simultaneous actions enables to simulate the effects of actions that are conceptually executed at the same time, but physically are performed separated in time, e.g., on a single or sequential processor system. In this section, we first explain the notion of simultaneous actions. Then, we show how simultaneous actions are supported in the framework. We illustrate the explanation with examples from the Packet-World.

B.3.1 Simultaneous Actions

In the literature, several researchers refer to simultaneously performed actions. Some examples: Allen and Ferguson [17] discuss "actions that interfere with each other" and that can have "additional synergistic effects". Boutilier and Brafman [46] mention "concurrent actions with a positive or negative interacting effect". Griffiths, Luck and d'Iverno [84] introduce the notions of "joint action that a group of agents perform together" and "concurrent actions, i.e. a set of actions performed at the same time". Joint actions and concurrent actions are based on the concepts of "strong and weak parallelism" introduced Kinny [113]. In [138], Michel, Gouaïch, and Ferber introduce the notions of weak and strong interactions that are related to simultaneous actions.

We denote simultaneous actions as actions that happen together and that can have a combined result. To calculate the effects of simultaneously performed actions that physically are performed separated in time, the actions are reified as influences [74]. Support for simultaneous actions requires two mechanisms: first, a mechanism is needed that determines which influences are treated as being executed together; second, a mechanism is needed that ensures that the combined outcome of simultaneously performed influences is in accordance with the domain that is modelled.

Determining Simultaneity. Simultaneity of influences is determined by a synchronization mechanism. Two possible mechanisms for synchronization are global synchronization and regional synchronization. With global synchronization, all agents in the multiagent system act simultaneously. Global synchronization is simple to implement, but the mechanism imposes centralized control. Regional synchronization offers more fine-grained synchronization. With regional synchronization, the composition of groups of synchronized agents—regions—depends on the actual locality of the agents and dynamically changes when agents enter or leave each others locality. Support for regional synchronization can be implemented as a service of the application environment [203, 59]. Alternatively, the agents can take care for the formation of regions themselves, providing a fully decentralized solution for synchronization. [207, 205] discuss a decentralized algorithm for regional synchronization in detail and provides a proof of correctness.

Imposing Domain Constraints. Domain constraints are imposed through a set of action laws. Action laws determine the effects of a set of synchronized influences on the state of the application environment. As such, action laws impose constrains on the implications of agents' (inter)actions. Fig. B.5 shows an example of simultaneous actions in the Packet-World. In the depicted situation, agents 3 can pass packets to agent 4 that can directly deliver the packets at the destination. Such packet transfer only succeeds when the two agents act together, i.e. agent 3 has to pass the packet while agent 4 simultaneously accepts the packet. To model

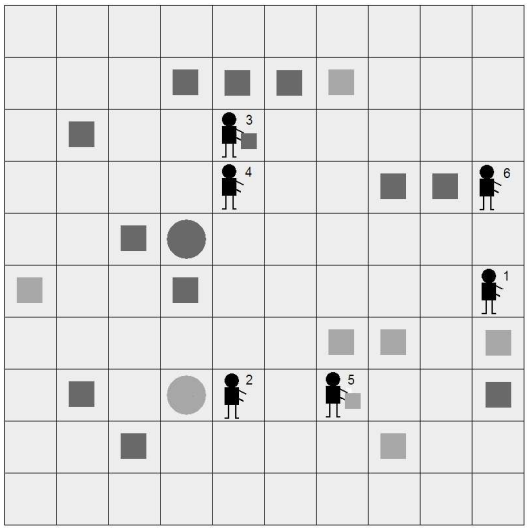


Figure B.5: Example of simultaneous actions in the Packet-World

the packet transfer, an action law is defined. This definition includes:

1. *The set of influences.* This set consists of two influences: `PassInfluence` and `AcceptInfluence`.
2. *The preconditions.* The packet transfer only succeeds if: (i) both agents have enough energy to execute the transfer, (ii) the locations of the agents match with a chain, (iii) the tail holds a packet and the head does not.
3. *The effects.* Applying the law properly reduces the energy level of both agents, and the packet is transferred from tail to head.

Notice that agent 2 and 5 also form a chain to transfer packets. In this chain however, packets are passed indirectly via the environment, i.e., agent 5 can put packets in between the two agents and agent 2 can pick the packets and deliver them at the destination. Contrary to the *synchronous* collaboration between agent 3 and 4, this *asynchronous* collaboration does not involve any simultaneous actions [204].

B.3.2 Support for Simultaneous Actions in the Framework

Fig. B.6 shows the main classes of the framework involved in the execution of simultaneous actions.

`Collector` collects the influences (`Influence`) invoked by the agents and stores

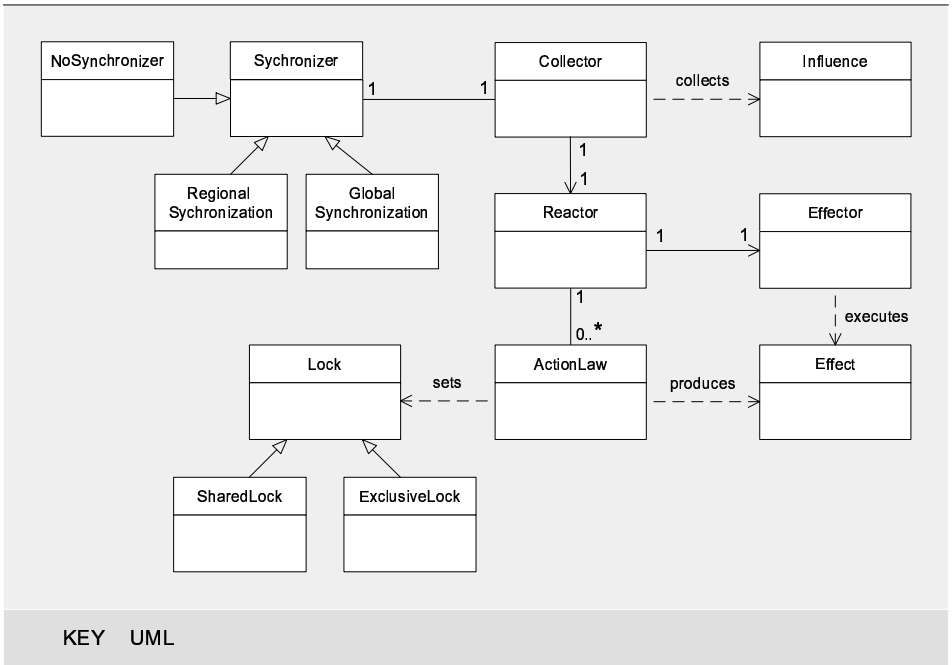


Figure B.6: Main classes of the framework involved in the execution of simultaneous actions

the influences in a buffer. Domain specific influences are defined as subclasses of `Influence`. A simple example is `StepInfluence` that is defined as follows:

```
public class StepInfluence extends Influence {
  private Direction direction;
  public StepInfluence(AgentId agent, Direction direction){
    super(agent);
    setDirection(direction);
  }
  ...
}
```

`Synchronizer` determines when influences are passed to the `Reactor` for execution. With `NoSychronizer` influences are passed one by one; with `GlobalSynchronizer` the set of influences of all agents is passed; with `RegionalSynchronization` the influences are passed to the `Reactor` per region. To form regions, the framework provides a default implementation for locality that is based on the default

range of perception. In particular, a region in the framework consists of the set of agents that are located within each other's perceptual range, or within the perceptual range of those agents, and so on. Applied to the situation in Fig. B.5: with `NOSynchronizer` all agents act asynchronously (in this case there is no support for passing packets directly from one agent to another); with `GlobalSynchronization` all agents act at one global pace; with `RegionalSynchronization` each agent act simultaneously with the other agents within its region. If we assume a perceptual range of two fields, than there are three regions: agents 3 and 4, agents 2 and 5, and agent 1 and 6. If in the depicted situation agent 1 makes a step towards South-West it enters the region of agents 5 and 2, while the original region of agent 1 and 6 is than reduced to only agent 6.

`Reactor` receives sets of influences from the `Collector` and calculates the effects (`Effect`), i.e., state changes in the application environment. Therefore, the reactor uses the sets of action laws (`ActionLaw`). The ordering in which laws are applied depends on the number of influences involved in the law. The law with the highest number of influences is applied first, then the law with the second highest number is applied, and so on. For laws with an equal number of influences, laws are applied in random order. Domain specific action laws have to be defined as subclasses of `ActionLaw`. This definition requires the implementation of four methods:

```
public <T extends Influence> Vector<Class<T>> getInfluenceTypes();
public boolean checkConditions(Vector<Influence> influences);
public Vector<Effect> getEffects(Vector<Influence> influences);
public Vector<Lock> getLocks();
```

The method `getInfluenceTypes()` returns a vector of influence types, one for each influence involved in the law. This method allows the reactor to check whether the law is applicable or not. For `TransferPacketLaw` that models the rules for agents to transfer a packet, `getInfluenceTypes()` is defined as follows:

```
public <T extends Influence> Vector<Class<T>> getInfluenceTypes(){
  Vector<Class<T>> infs = new Vector<Class<T>>();
  infs.add((Class<T>)PassInfluence.class);
  infs.add((Class<T>)AcceptInfluence.class);
  return infs;
}
```

The method `checkConditions()` verifies whether the necessary conditions hold to apply the law. For `TransferPacketLaw` the conditions are:

```
public boolean checkConditions((Vector<Influence> infs)){
```

```
    //the agents must have energy
    passAgent = (AgentState)getState().getItem(infs[1].getAgent());
    if (passAgent.getEnergyLevel() <= 0)
      return false;
    ...
    //passAgent must hold a packet, acceptAgent not
    boolean hold = false;
    for (Relation rel :  getState().getRelations(HoldRelation.class)){
     if (rel.containsItem(passAgent)){
       hold = true;
       break;
     }
    }
    ...
}
```

The method `getEffects()` returns the effects induced by the law. An application specific effect has to be defined as a subclass of `Effect`. A simple example is `AddRelationEffect` that is used to add a relation in the state of the application environment. Such relation is used to link the agent with the packet it accepts during a packet transfer:

```
public class AddRelationEffect extends Effect {
  public AddRelationEffect(GridState state, Relation relation){
    super(state);
    setRelation(relation);
  }
  public void execute(){
    state.addRelation(relation);
  }
  ...
}
```

Finally, each action law has to implement the method `getLocks()`. This method returns the locks on the state elements used by the law. Locks (`Lock`) avoid conflicts between action laws. To ensure that all simultaneously performed influences are applied in the same circumstances, the action laws produce the effects of influences from the same state of the application environment. However, applying a law may induce constraints on state elements. For example, assume that `StepLaw` handles the movement of a single agent. If agent 6 in Fig. B.5 makes a step to South than agent 1 can no longer step to North. To avoid a conflict between the application of the law for both agents, the first application of `StepLaw` puts a lock on the field the agent moves to. During the execution of the law for the other influence

of the region, the reactor uses the lock to check whether the `StepLaw` is applicable or not. The framework supports two types of basic locks: `ExclusiveLock` and `SharedLock`. An `ExclusiveLock` on a state element of the application environment excludes other laws to access the locked element. A `SharedLock` allows other laws to put a shared lock on the element, however, it excludes a possible `ExclusiveLock`.

`Effector` is responsible to apply the effects induced by the action laws. Each `Effect` implements the method `execute()` that actually performs the effects to the state of the application environment, see the `AddRelationEffect` above.

Hot Spots. Much of the complexity to deal with simultaneous actions is hidden by the framework core. If the application requires support for simultaneous actions, the developer has to select a particular type of synchronization. This selection has to be specified in the `EnvironmentFactory` definition. Furthermore, the developer has to define application specific instances for `Influence`, `ActionLaw`, `Lock`, and `Effect`, as illustrated above.

B.4 Applying the Framework to an Experimental Robot Application

As a validation, we have used the framework to develop an experimental warehouse application in which two Lego-Mindstorms robots collaborate to organize the supply of products. In this section we give an overview of application. First, we introduce the application based on a simulation, and we explain the setup of the physical system with robots and the environment. Then we show how we have instantiated the framework to develop the application software. We limit the discussion to the agent software.

B.4.1 Robot Application

Fig. B.7 shows a schematical overview of the robot application.

The environment consists of two zones: the corridor on the left side in which a non-mobile crane robot can manoeuvre, and the rectangular factory flour on the right side in which a mobile robot can move around. The colored packets on the right side of the factory floor represent products. The circle represents the delivering point for products. The task of the robots is to guarantee a stream of products from supply to drainage.

We have developed the robots with the Lego-Mindstorms packet [10]. Besides building blocks to construct robots, Lego-Mindstorms offers a programmable microcomputer called Robotic Command eXplorer (RCX) to program a robot. To enable the robot to interact with the environment, various sensors (light, pressure,

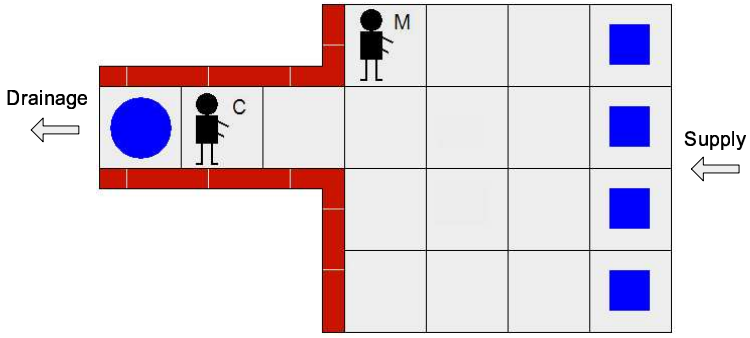


Figure B.7: Simulation of the robot application

etc.) and actuators (switches, motors, etc.) are available that can be connected to the RCX. Furthermore, the RCX is equipped with an infrared serial communication interface that enables a developer to program the microcomputer. We have used the LeJOS (Lego Java Operating System), as a replacement firmware for the Lego Mindstorms RCX. LeJOS is a reduced Java Virtual Machine that fits within the 32kb on the RCX, and that allows to program a Lego robot with Java [11].

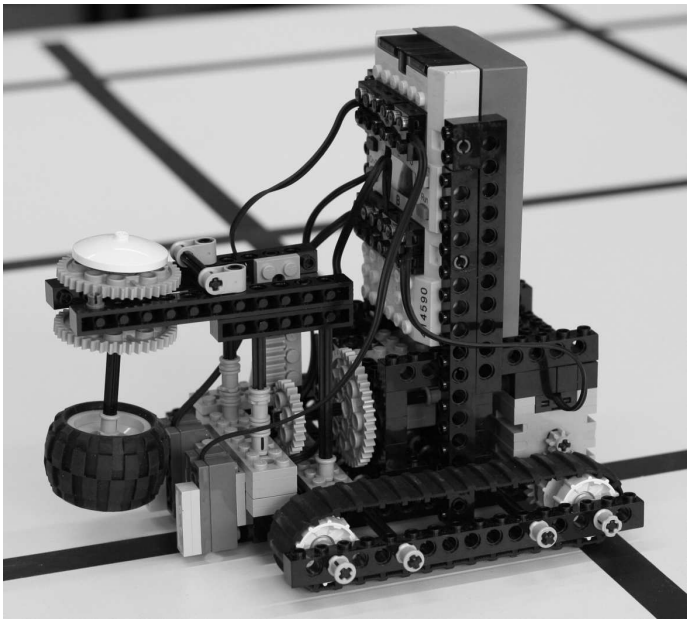


Figure B.8: A robot carrying a packet

Fig. B.8 shows one of the robots. Robots are equipped with various sensors to monitor the environment, and they have two grasp arms to pick up packets. The robots can communicate with a local computer via infrared communication.

Fig. B.9 shows environment with the two robots in action. The robots use light sensors to follow the paths that are marked by black lines.

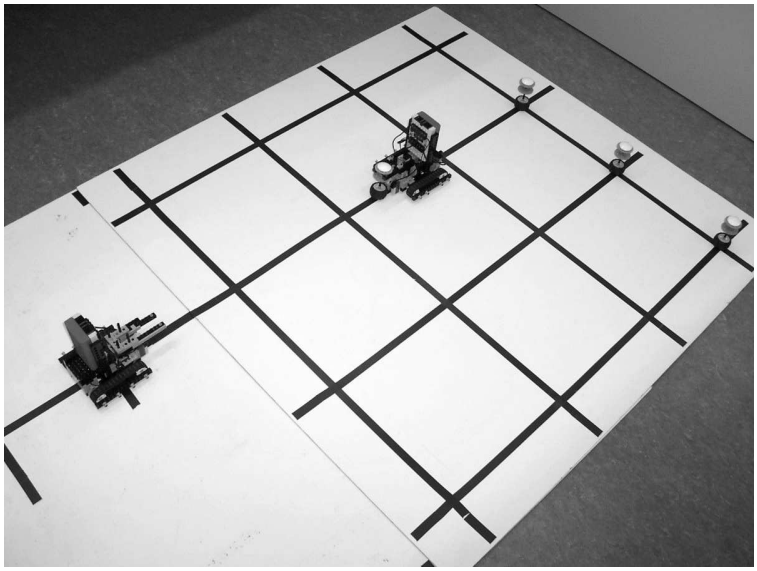


Figure B.9: The environment with the robots in action

B.4.2 Applying the Framework

Due to memory limitations of the RCX, it was not possible to execute the robot control software directly on the robot hardware. Therefore the robot software is divided in two collaborating programs: one program running on the RCX of the robot that monitors the environment and executes actions, and a second program running on a local computer that selects actions. Fig. B.10 shows how the robot software is deployed on the various hardware units. The agents use `LTDSchedule` as scheduling schema. `LTDSchedule` is a predefined scheduling schema in the framework that successively activates perception, communication, decision making in an endless loop. `Perception` transforms the data sensed by `InfraredSensor` into a percept (`WAStatePercept`). Periodically, the RCX sends an infrared message with the current status of the robot (position, hold packet or not) to the agent program on the computer. Infrared communication is handled by `IRTower` and `IRPort` on the host computer and the RCX respectively. The decision making mod-

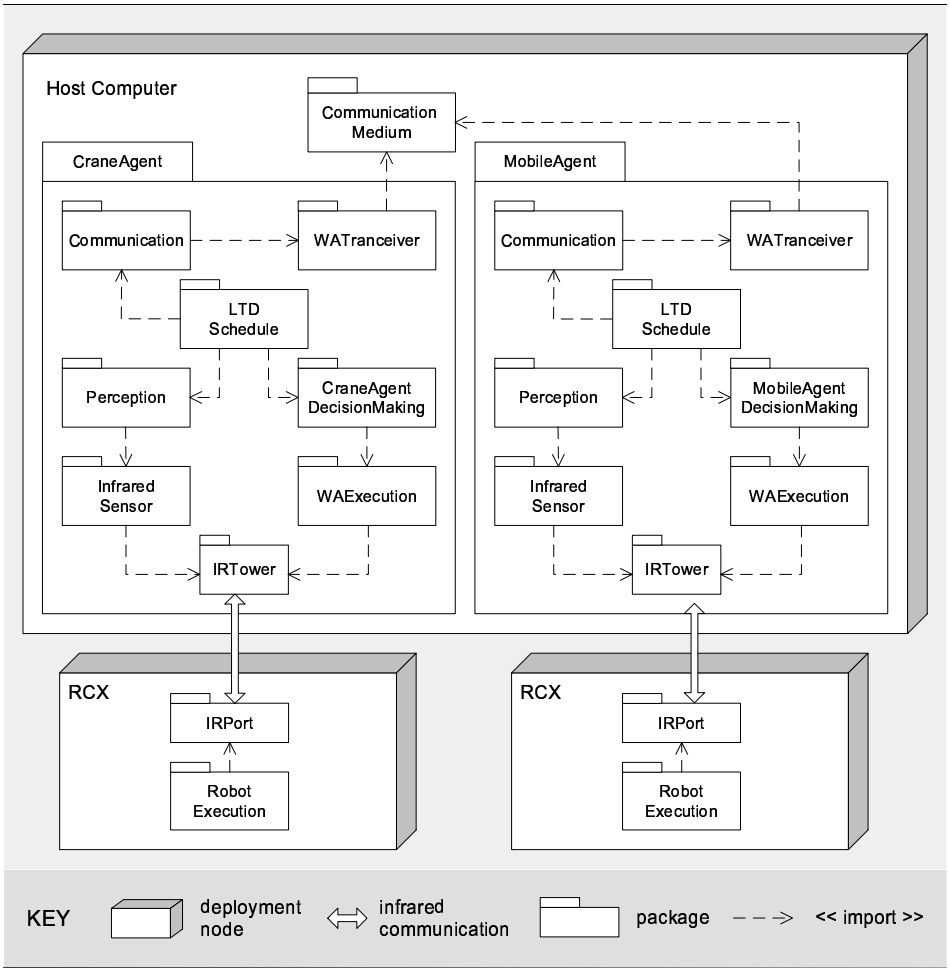


Figure B.10: Deployment of the robot software

ules (`CraneAgentDecisionMaking` and `MobileAgentDecisionMaking`) take care for action selection. The action selection mechanisms of the `MobileAgent` continuously executes a sequence of three roles: `LookForPacket`, `ReturnToCorridor`, and `PassPacket`. When the `MobileAgent` arrives with a new packet at the corridor, the `CraneAgent` executes `AcceptPacket`, it delivers the packet (`DeliverPacket`) at the destination, and subsequently waits for the next packet (`Wait`). `WAExecution` sends the selected actions to the RCX via the `IRTower`. The decision making modules however, produce high-level actions, such as “drive to the corridor” and

"put packet on the destination". When `RobotExecution` receives such an action, it translates the actions into low-level actions to steer the actuators.

When the `MobileAgent` arrives with a packet at the corridor, it has to pass the packet to the `CraneAgent`. To coordinate this interaction, the agents use a `PassPacketProtocol` that is handled by the `Communication` module. The subsequent steps of this protocol are depicted in Fig. B.11. When the `MobileAgent`

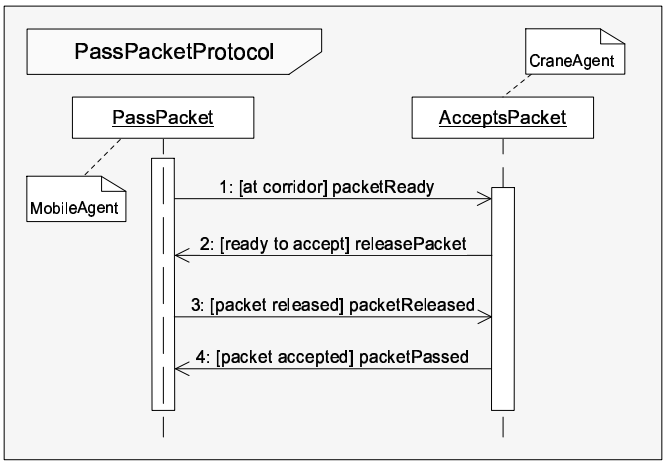


Figure B.11: Protocol to coordinate the transfer of a packet

arrives at the corridor it informs the `CraneAgent` that it has arrived with a packet for delivering. The `CraneAgent` drives towards the packet and as soon as it is in the correct position, it informs the `MobileAgent` to release the packet. When the `MobileAgent` has released the packet it informs the `CraneAgent`. This latter then brings the packet to the delivering location. To communicate with one another, robots use `WATransceiver` that transmits the messages via a simple `CommunicationMedium`.

Simulation of the Robot Application. We also developed a simulation of the robot application (see Fig. B.7). In the simulated environment, the agents step on a grid. Products are represented by packets, and the drainage by a delivering point, similar as in the Packet-World. For the simulation, we were able to fully reuse the implementations of the agents, including the hot spots for perception, decision making, and the communication protocol for coordinating the packet transfer. The software agents also use the same scheduling schema as the robots. InfraRedSensor, WAExecution, and WATransceiver were no longer necessary for the agents in the software environment. The functionality for interfacing with the

software environment is managed by the framework core.

List of Publications Danny Weyns

Articles in international reviewed journals

1. D. Weyns, A. Omicini, and J. Odell, Environment: First-class abstraction in multiagent systems, Special Issue on Environments in Multiagent Systems, Journal on Autonomous Agents and Multiagent Systems, 14(1), 2007 (to appear).
2. H.V.D. Parunak, and D. Weyns, Environments in multiagent systems, Guest Editorial, Special Issue on Environments in Multiagent Systems, Journal on Autonomous Agents and Multiagent Systems, 14(1), 2007 (to appear).
3. K. Schelfthout, D. Weyns, T. Holvoet, Middleware that enables protocol-based coordination applied in automatic guided vehicle control, IEEE Distributed Systems on Line, 7(3), August, 2006.
4. D. Weyns, and T. Holvoet, On the role of environments in multiagent systems, Informatica 29 (4), pp. 409-421, 2005.
5. D. Weyns, M. Schumacher, A. Ricci, M. Viroli, and T. Holvoet, Environments in multiagent systems, The Knowledge Engineering Review 20 (2), pp. 127-141, June, 2005.
6. D. Weyns, E. Truyen, and P. Verbaeten, Serialization of Distributed Threads in Java, Parallel and Distributed Computing Practices 6 (1), pp. 81-98, July, 2005.
7. D. Weyns, and T. Holvoet, On environments in multi-agent systems, AgentLink Newsletter 16 , pp. 18-19, December, 2004.
8. D. Weyns, E. Steegmans, and T. Holvoet, Towards active perception in situated multi-agent systems, Applied Artificial Intelligence 18 (9-10), pp. 867-883, October, 2004.
9. D. Weyns, and T. Holvoet, A formal model for situated multi-agent systems, Fundamenta Informaticae 63 (2-3), pp. 125-158, November, 2004.

Edited Volumes

1. H.V.D. Parunak, and D. Weyns, Environments in Multiagent Systems (eds.), Special Issue on Environments in Multiagent Systems, Journal on Autonomous Agents and Multiagent Systems, 14(1), 2007 (to appear).
2. D. Weyns, and T. Holvoet, (eds.), Multiagent Systems and Software Architecture, Proceedings of the Special Track on Multiagent Systems and Software Architecture at Net.ObjectDays, 2006.

3. D. Weyns, H.V.D Parunak and F. Michel, (eds.), Environments for Multiagent Systems III, Post-proceedings of the Third International Workshop on Environments for Multiagent Systems, Springer-Verlag, Lecture Notes in Computer Science Series, 2007, (to appear).

4. D. Weyns, H.V.D. Parunak and F. Michel, (eds.), Environments for Multiagent Systems II, Post-proceedings of the Second International Workshop on Environments for Multiagent Systems, Springer-Verlag, Lecture Notes in Computer Science Series, Vol. 3830, 2006.

5. D. Weyns, H.V.D. Parunak and F. Michel, (eds.), Environments for Multiagent Systems, Post-proceedings of the First International Workshop on Environments for Multiagent Systems, Springer-Verlag, Lecture Notes in Computer Science Series, Vol. 3374, 2005.

Contributions at international conferences, published in proceedings

1. D. Weyns, and T. Holvoet, Multiagent systems and software architecture, Proceedings of the Special Track on Multiagent Systems and Software Architecture at Net.ObjectDays, 2006.

2. N. Boucké, D. Weyns, and T. Holvoet, Experiences with Theme/UML for architectural design of a multiagent system, Proceedings of the Special Track on Multiagent Systems and Software Architecture at Net.ObjectDays, 2006.

3. D. Weyns, and T. Holvoet, On the connection between multiagent systems and software architecture, Invited contribution at the 7th International Workshop on Engineering Societies of the Agents World, 2006 (to appear).

4. D. Weyns, N. Boucké, and T. Holvoet, Gradient field-based task assignment in an AGV transportation system, Proceedings of the 5th International Conference on Autonomous Agents and Multiagent Systems, (Stone, P. and Weiss, G., eds.), pp. 842-849, 2006.

5. D. Weyns, and T. Holvoet, Multiagent systems as software architecture: Another perspective on software engineering with multiagent systems, Proceedings of the 5th International Conference on Autonomous Agents and Multiagent Systems, (Stone, P. and Weiss, G., eds.), pp. 1344-1347, 2006.

6. D. Weyns, and T. Holvoet, A reference architecture for situated multiagent systems, Proceedings of the 3th International Workshop on Environments for Multiagent Systems, (Weyns, D. and Parunak, V. and Michel, F.), pp. 120-170, 2006.

7. D. Weyns and T. Holvoet, Architectural design of an industrial AGV transportation system with a multiagent system approach, Software Architecture Technology User Network Workshop, SATURN, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, 2006 (to appear).

8. N. Boucké, D. Weyns, K. Schelfthout, and T. Holvoet, Applying the ATAM to an architecture for decentralized control of a transportation system, Proceedings of the 2nd International Conference on Quality of Software Architecture, Vasteras, Sweden, Lecture Notes in Computer Science, 2006 (to appear).

9. D. Weyns, A. Helleboogh, and T. Holvoet, The Packet-World: A testbed for investigating situated multiagent systems, Software Agent-Based Applications, Platforms, and Development Kits, (Unland, R. and Klush, M. and Calisti, M., eds.), Whitestein Series in Software Agent Technologies, Birkhauser Verlag, Basel - Boston - Berlin, September, pp. 383-408, 2005.

10. D. Weyns, K. Schelfthout, and T. Holvoet, Exploiting a virtual environment in a real-world application, Environments for Multiagent Systems II (Weyns, D. and Parunak, V. and Michel, F., eds.), vol 3830, Lecture Notes in Computer Science, pp. 218-234, 2006

11. D. Weyns, G. Vizzari, and T. Holvoet, Environments for situated multiagent systems: Beyond infrastructure, Environments for Multi-Agent Systems II (Weyns, D. and Parunak, V. and Michel, F., eds.), vol 3830, Lecture Notes in Computer Science, pp. 1-17, 2006

12. D. Weyns, K. Schelfthout, and T. Holvoet, Architectural design of a distributed application with autonomic quality requirements, Proceedings Design and Evolution of Autonomic application Software (Garlan, D. and Litoui, M. and Mller, H. and Mylopoulos, J. and Smith, D. and Wong, K., eds.), pp. 52-59, 2005

13. D. Weyns, K. Schelfthout, T. Holvoet, and O. Glorieux, Towards adaptive role selection for behavior-based agents, Adaptive Agents and Multi-Agent Systems III: Adaptation and Multi-Agent Learning (Kudenko, D. and Kazakov, D. and Alonso, E., eds.), vol 3394, Lecture Notes in Computer Science, pp. 295-314, 2005

14. D. Weyns, E. Steegmans, and T. Holvoet, Integrating free-flow architectures with role models based on statecharts, Software Engineering for Multi-Agent Systems III: Research Issues and Practical Applications (Choren, R. and Garcia, A. and Lucena, C. et al., eds.), vol 3390, Lecture Notes in Computer Science, pp. 104-120, 2005

15. D. Weyns, G. Vizzari, and T. Holvoet, Environments for multiagent systems: Beyond infrastructure, Environments for Multiagent Systems (Weyns, D. and Parunak, V. and Michel, F., eds.), pp. 101-117, 2005

16. D. Weyns, H. Parunak, F. Michel, T. Holvoet, and J. Ferber, Environments for multi-agent systems, state-of-the-art and research challenges, Environments for multi-agent systems (Weyns, D. and Parunak, H.V.D. and Michel, F., eds.), vol 3374, Lecture Notes in Computer Science, pp. 1-48, 2005

17. D. Weyns, K. Schelfthout, and T. Holvoet, Exploiting a virtual environment in a real-world application, Environments for Multiagent Systems (Weyns, D. and Parunak, V. and Michel, F., eds.), pp. 21-36, 2005

18. D. Weyns, N. Boucké, T. Holvoet, and W. Schols, Gradient field-based task assignment in an AGV transportation system, Proceedings of Third European Workshop on Multiagent Systems (Gleizes, M.P. and Kaminka, G. and Now, A. and Ossowski, S. and Tuyls, K. and Verbeeck, K., eds.), pp. 447-459, 2005

19. D. Weyns, and T. Holvoet, From reactive robotics to situated multiagent systems: a historical perspective on the role of the environment in multiagent systems, Sixth International Workshop on Engineering Societies in the Agents World (Dikenelli, O. and Gleizes, M.P. and Ricci, A., eds.), pp. 31-56, 2005

20. D. Weyns, K. Schelfthout, T. Holvoet, T. Lefever, and J. Wielemans, Architecture-centric development of an AGV transportation system, Multi-Agent Systems and Applications IV (Pechoucek, M. and Petta, P. and Varga, L.Z., eds.), vol 3690, Lecture Notes in Computer Science, pp. 640-645, 2005

21. D. Weyns, K. Schelfthout, T. Holvoet, and T. Lefever, Decentralized control of E'GV transportation systems, Autonomous Agents and Multiagent Systems, Industry Track (Pechoucek, M. and Steiner, D. and Thompson, S., eds.), pp. 67-74, 2005

22. E. Steegmans, D. Weyns, T. Holvoet, and Y. Berbers, A design process for adaptive behavior of situated agents, Agent-Oriented Software Engineering V (Odell, J. and Giorgini, P. and Mller, J.P., eds.), vol 3382, Lecture Notes in Computer Science, pp. 109-125, 2005

23. K. Schelfthout, D. Weyns, and T. Holvoet, Middleware for protocol-based coordination in dynamic networks, Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing (Terzis, S. and Donsez, D., eds.), pp. 1-8, 2005

24. T. Holvoet, D. Weyns, K. Schelfthout, and T. Lefever, Decentralized control of autonomous guided vehicles scalable warehouse systems, The IEE Seminar on Autonomous Agents, ISSN 0537-9989, pp. 11-18, 2005

25. A. Helleboogh, T. Holvoet, D. Weyns, and Y. Berbers, Extending time management support for multi-agent systems, Multi-Agent and Multi-Agent-Based Simulation: Joint Workshop MABS 2004, New York, NY, USA, July 19, 2004, Revised Selected Papers (Davidsson, P. and Logan, B. and Takadama, K., eds.), vol 3415, Lecture Notes in Computer Science, pp. 37-48, 2005

26. A. Helleboogh, T. Holvoet, D. Weyns, and Y. Berbers, Towards time management adaptability in multi-agent systems, Adaptive Agents and Multi-Agent Systems III: Adaptation and Multi-Agent Learning (Kudenko, D. and Kazakov, D. and Alonso, E., eds.), vol 3394, Lecture Notes in Computer Science, pp. 88-105, 2005

27. D. Weyns, E. Steegmans, and T. Holvoet, Protocol-based communication for situated agents, Proceedings of the Sixteenth Belgium-Netherlands Conference on Artificial Intelligence (Verbrugge, R. and Taatgen, N. and Schomaker, L., eds.), pp. 303-304, 2004

28. D. Weyns, E. Steegmans, and T. Holvoet, Towards commitments for situated agents, IEEE Special Track on Agents and Roles, SMC'2004 Conference Proceedings (Thissen, W. and Wieringa, P. and Pantic, M. and Ludema, M., eds.), pp. 5479-5485, 2004

29. D. Weyns, E. Steegmans, and T. Holvoet, Protocol based communication for situated multi-agent systems, Proceedings of The Third International Joint Conference on Autonomous Agents and Multi Agent Systems (Jennings, N. and Sierra, C. and Sonenberg, L. and Tambe, M., eds.), pp. 118-126, 2004

30. D. Weyns, and T. Holvoet, Situated multi-agent systems for developing self-managing distributed applications, Proceedings of the Doctoral Mentoring Symposium, Third International Joint Conference on Autonomous Agents and Multiagent Systems, 2004 (Kaminka, G., ed.), pp. 25-26, 2004

31. D. Weyns, E. Steegmans, and T. Holvoet, Combining adaptive behavior and role modeling with state charts, Proceedings of the Third International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (Choren, R. and Garcia, A. and Lucena, C. and Griss, M. and Kung, D. and Minsky, N. and Romanovsky, A., eds.), pp. 81-90, 2004

32. D. Weyns, K. Schelfthout, T. Holvoet, and O. Glorieux, A role based model for adaptive agents, Proceedings of the AISB 2004, Fourth Symposium on Adaptive Agents and Multi-Agent Systems, pp. 75-86, 2004

33. D. Weyns, and T. Holvoet, A colored Petri net for regional synchronization in situated multiagent systems, Proceedings of the First International Workshop on Coordination and Petri Nets (Ciancarini, P. and Bocchi, L., eds.), pp. 65-86, 2004

34. D. Weyns, A. Helleboogh, E. Steegmans, T. De Wolf, K. Mertens, N. Boucké, and T. Holvoet, Agents are not part of the problem, agents can solve the problem, Proceedings of the OOPSLA Workshop on Agent-Oriented Methodologies, 2004 (Gonzales-Perez, C., ed.), pp. 101-112, 2004

35. E. Steegmans, D. Weyns, T. Holvoet, and Y. Berbers, Designing roles for situated agents, The fifth international workshop on agent-oriented software engineering (Odell, J. and Giorgini, P. and Muller, J.P., eds.), pp. 17-32, 2004

36. A. Helleboogh, T. Holvoet, and D. Weyns, Time management adaptability in multi-agent systems, Proceedings of the AISB 2004 Fourth Symposium on Adaptive Agents and Multi-Agent Systems (Kudenko, D. and Alonso, E. and Kazakov, D., eds.), pp. 20-30, 2004

37. A. Helleboogh, T. Holvoet, and D. Weyns, Time management support for simulating multi-agent systems, Joint workshop on multi-agent and multi-agent-based simulation (Davidsson, P. and Gasser, L. and Logan, B. and Takadama, K., eds.), pp. 31-40, 2004

38. N. Boucké, D. Weyns, T. Holvoet, and K. Mertens, Decentralized allocation of tasks with delayed commencement, EUMAS'04 Proceedings (Chiara, G. and Giorgini, P. and van der Hoek, W., eds.), pp. 57-68, 2004

39. D. Weyns, and T. Holvoet, Regional synchronization for simultaneous actions in situated multi-agent systems, Multi-Agent Systems and Applications III (Marik, V. and Mller, J. and Pechoucek, M., eds.), vol. 2691, Lecture Notes in Computer Science, Lecture Notes in Computer Science, pp. 497-511, 2003

40. D. Weyns, E. Truyen, and P. Verbaeten, Serialization of distributed execution-state in Java, Objects, Components, Architectures, Services, and Applications for a Networked World (Aksit, M. and Mezini, M. and Unland, R., eds.), vol 2591, Lecture Notes in Computer Science, pp. 41-61, 2003

41. D. Weyns, and T. Holvoet, Model for situated multi-agent systems with regional synchronization, Concurrent Engineering, Enhanced Interoperable Systems (Jardim-Goncalvas, R. and Cha, J. and Steiger-Garcao, A., eds.), pp. 177-188, 2003

42. D. Weyns, and T. Holvoet, Synchronous versus asynchronous collaboration in situated multi-agent systems, Proceedings of the Second International

Joint Conference on Autonomous Agents and Multiagent Systems (Rosenschein, J. and Sandholm, T. and Wooldridge, M. and Yokoo, M., eds.), pp. 1156-1158, 2003

43. D. Weyns, and T. Holvoet, Model for simultaneous actions in situated multi-agent systems, Multiagent System Technologies (Schillo, M. and Klusch, M. and Muller, J. and Tianfield, H., eds.), vol 2831, Lecture Notes in Computer Science, pp. 105-119, 2003

44. D. Weyns, E. Steegmans, and T. Holvoet, A model for active perception in situated multi-agent systems, Proceedings of the First European Workshop on Multi-Agent Systems (d'Iverno, M. and Sierra, C. and Zambonelli, F., eds.), pp. 1-15, 2003

45. D. Weyns, E. Truyen, and P. Verbaeten, Serialization of a Distributed Execution-state in Java, Hauptkonferenz Net.ObjectDays 2002 (Aksit, M. and Mezini, M., eds.), pp. 55-72, 2002

46. D. Weyns, E. Truyen, and P. Verbaeten, Distributed Threads in Java, Informatica (Grigoras, D., ed.), pp. 94-109, 2002

47. D. Weyns, and T. Holvoet, Look, talk and do: a synchronization scheme for situated multi-agent systems, Proceedings of UKMAS '02 (McBurney, P. and Wooldridge, M., eds.), pp. 1-8, 2002

48. D. Weyns, and T. Holvoet, A colored Petri-net for a multi-agent application, Proceedings of MOCA'02 (Moldt, D., ed.), vol 561, DAIMI PB, pp. 121-141, 2002

49. K. Schelfthout, T. Coninx, A. Helleboogh, T. Holvoet, E. Steegmans, and D. Weyns, Agent Implementation Patterns, Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies (Debenham, J. and Henderson-Sellers, B. and Jennings, N. and Odell, J., eds.), pp. 119-130, 2002

Contributions at international conferences, not published or only as abstract

1. D. Weyns, and T. Holvoet, The Packet-World as a case to study Sociality in Multi-Agent Systems, Autonomous Agents and Multi-Agent Systems, AAMAS 2002, Bolgona, Italy, July 15-19, 2002, Universite di Bologna and Universite di Modena e Reggio Emilia

2. T. Holvoet and D. Weyns, Environments in Multiagent Systems, Tutorial at the Eighth European Agent Systems Summer School, EASSS, Annecy, France, 2006.

Technical reports

1. W. Schols, N. Boucké, D. Weyns, and T. Holvoet, Gradient field based order assignment in AGV systems, Department of Computer Science, K.U.Leuven, Report CW 425, Leuven, Belgium, September, 2005

2. N. Boucké, T. Holvoet, T. Lefever, R. Sempels, K. Schelfthout, D. Weyns, and J. Wielemans, Applying the architecture tradeoff analysis method (ATAM) to an industrial multi-agent system application, Department of Computer Science, K.U.Leuven, Report CW 431, Leuven, Belgium, December, 2005

Curriculum vitae

Danny Weyns was born in Aarschot (Belgium) on September 30, 1958. After he received the degree of Industrieel Ingenieur from the Hoger Instituut Kempen in 1980, he worked as a lector at the Hogeschool voor Wetenschap en Kunst in Brussels. In 2001, Danny received a Master's degree in Informatics (Aanvullende Opleiding Toegepaste Informatica) from the K.U. Leuven. He graduated summa cum laude with the thesis "Serialization of Distributed Execution State in Java" under the supervision of Prof. Pierre Verbaeten. The same year, he started working as a research assistant in the DistriNet (Distributed systems and computer Networks) research group at the Department of Computer Science at the K.U.Leuven. From 2004-2006, he participated in a joint IWT project with Egemin on the development of a decentralized control architecture for automated guided vehicles. Danny was co-organizer of the series of workshops on environments in multiagent systems that were held in New York 2004, Utrecht 2005, and Hakodate 2006. He was also the chair of the AgentLink technical forum group on environments in multiagent systems in 2005 in Ljubljana and Budapest. Danny served on the Program Commitee of various international conferences, and he performed review work for several journals.

Een architectuur-gebaseerde aanpak voor software ontwikkeling met gesitueerde multiagent systemen

Nederlandse samenvatting

Beknopte samenvatting

De ontwikkeling en het beheer van hedendaagse gedistribueerde software toepassingen is moeilijk. Drie belangrijke redenen voor de toenemende complexiteit zijn: (1) belanghebbenden van de software hebben verschillende, vaak tegenstrijdige, kwaliteitsvereisten voor de systemen; (2) de software systemen dienen om te gaan met voortdurende wijzigingen in hun omgeving; (3) activiteit in de systemen is inherent lokaal, globale controle is moeilijk te verwezenlijken of helemaal uitgesloten.

In deze thesis stellen we een aanpak voor om dergelijke complexe systemen te ontwikkelen. Deze aanpak integreert gesitueerde multiagent systemen als software architectuur in een algemeen software ontwikkelingsproces. Sleutelaspecten van de aanpak zijn architectuur-gebaseerde software ontwikkeling, zelfbeheer en decentrale controle. Architectuur-gebaseerde software ontwikkeling zet de belanghebbenden van een software systeem aan om expliciet om te gaan met tegenstrijdige kwaliteitsdoelen. Zelfbeheer laat toe dat een software systeem zelfstandig omgaat met voortdurende wijzigingen in de omgeving. Decentrale controle biedt een antwoord op de inherente localiteit van activiteit. In een systeem waar globale controle geen optie is, dient de vereiste functionaliteit gerealiseerd te worden door samenwerkende deelsystemen.

Tijdens ons onderzoek hebben we een geavanceerd model ontwikkeld voor gesitueerde multiagent systemen. Dit model integreert gesitueerde agenten en de omgeving als expliciete ontwerpabstracties in een gesitueerd multiagent systeem. Het model voorziet in een aantal geavanceerde mechanismen voor adaptief gedrag. Deze mechanismen laten toe dat gesitueerde agenten zich snel kunnen aanpassen aan wijzigende omstandigheden in het systeem. Agenten kunnen het systeem verlaten of nieuwe agenten kunnen worden toegevoegd zonder de rest van het systeem te verstoren. Controle in een gesitueerd multiagent systeem is gedecentraliseerd, de agenten werken samen om de systeemfunctionaliteit te realiseren.

Uit ervaring met het ontwikkelen van verschillende gesitueerde multiagent systeem applicaties hebben we een referentiearchitectuur ontwikkeld voor gesitueerde multiagent systemen. Deze referentiearchitectuur mapt het geavanceerde model voor gesitueerde multiagent systemen op een abstracte systeemdecompositie. De referentiearchitectuur kan worden toegepast bij de ontwikkeling van nieuwe toepassingen met soortgelijke kenmerken en vereisten.

Tijdens ons onderzoek hebben we een gesitueerd multiagent systeem toegepast in een industrieel transportsysteem met automatisch bestuurde voertuigen. Het ontwerp en de evaluatie van de software architectuur van deze applicatie hebben in belangrijke mate bijgedragen tot de ontwikkeling van de referentiearchitectuur. De succesvolle ontwikkeling van deze complexe toepassing toont aan hoe multiagent systemen als software architectuur kunnen geïntegreerd worden in een algemeen software ontwikkelingsproces.

1 Inleiding

De ontwikkeling en het beheer van hedendaagse gedistribueerde software toepassingen is moeilijk. Drie belangrijke redenen voor de toenemende complexiteit vormen het startpunt van dit onderzoek. Een eerste reden is de toenemende vraag naar de kwaliteit van de software [3, 20]. Belanghebbenden van de software (projectleiders, gebruikers, architecten, programmeurs, e.d.) hebben verschillende, vaak tegenstrijdige, kwaliteitsvereisten voor de systemen (gebruiksvriendelijkheid, veiligheid, aanpasbaarheid e.d.). Het vinden van de juiste balans tussen de vereisten, en de ontwikkeling van software die voldoet aan de belangrijkste vereisten is een moeilijke opdracht voor de software ontwikkelaars.

Een tweede belangrijke reden waarom de ontwikkeling en het beheer van gedistribueerde toepassingen moeilijk is zijn de voortdurende wijzigingen en dynamiek waaraan software systemen onderhevig zijn. Voorbeelden zijn voortdurend wijzigende belastingen, veranderingen in de beschikbaarheid van diensten, en wijzigingen in de topologie van het netwerk [25, 19, 43].

Naast de complexiteit als gevolg van de kwaliteitsvereisten en de dynamische uitvoeringsomgeving wordt een belangrijke familie van gedistribueerde applicaties gekenmerkt door lokaliteit van activiteit. In dergelijke applicaties is globale toegang tot hulpbronnen en diensten moeilijk te bereiken of zelfs uitgesloten. Voorbeeld applicaties zijn geautomatiseerde transportsystemen, mobiele netwerken, en draadloze sensornetwerken.

In ons onderzoek stellen we een architectuur-gebaseerde aanpak voor om dergelijk complexe systemen te ontwikkelen. Architectuur–in het bijzonder software architectuur–is cruciaal voor het beheersen van complexiteit en het bereiken van de belangrijke kwaliteitsvereisten. De voorgestelde aanpak beoogt de ontwikkeling van systemen die in staat zijn zelfstandig om te gaan met de dynamische uitvoeringsomstandigheden. Centraal in de aanpak zijn gesitueerde multiagent systemen. Tijdens ons onderzoek hebben we een geavanceerd model ontwikkeld voor gesitueerde multiagent systemen. Dit model integreert gesitueerde agenten en de omgeving als expliciete ontwerpabstracties in een gesitueerd multiagent systeem. Het model voorziet in een aantal geavanceerde mechanismen voor adaptief gedrag. Deze mechanismen laten toe dat gesitueerde agenten zich snel kunnen aanpassen aan wijzigende omstandigheden in het systeem. Agenten kunnen het systeem verlaten of nieuwe agenten kunnen worden toegevoegd zonder de rest van het systeem te verstoren. Controle in een gesitueerd multiagent systeem is gedecentraliseerd, de agenten werken samen om de systeemfunctionaliteit te realiseren.

Uit ervaring met het ontwikkelen van verschillende gesitueerde multiagent systeem applicaties hebben we een referentiearchitectuur ontwikkeld voor gesitueerde multiagent systemen. Deze referentiearchitectuur beeldt het geavanceerde model voor gesitueerde multiagent systemen af op een abstracte systeemdecompositie. De referentiearchitectuur is beschreven aan de hand van verschillende invalshoeken. De referentiearchitectuur integreert een aantal patronen die hergebruikt kunnen

worden tijdens de ontwikkeling van nieuwe toepassingen.

Tijdens ons onderzoek hebben we een gesitueerd multiagent systeem toegepast in een industrieel transportsysteem met automatisch bestuurde voertuigen. Het ontwerp en de evaluatie van de software architectuur van deze applicatie hebben in belangrijke mate bijgedragen tot de ontwikkeling van de referentiearchitectuur. De succesvolle ontwikkeling van deze complexe toepassing toont aan hoe multiagent systemen als software architectuur kunnen geïntegreerd worden in een algemeen software ontwikkelingsproces.

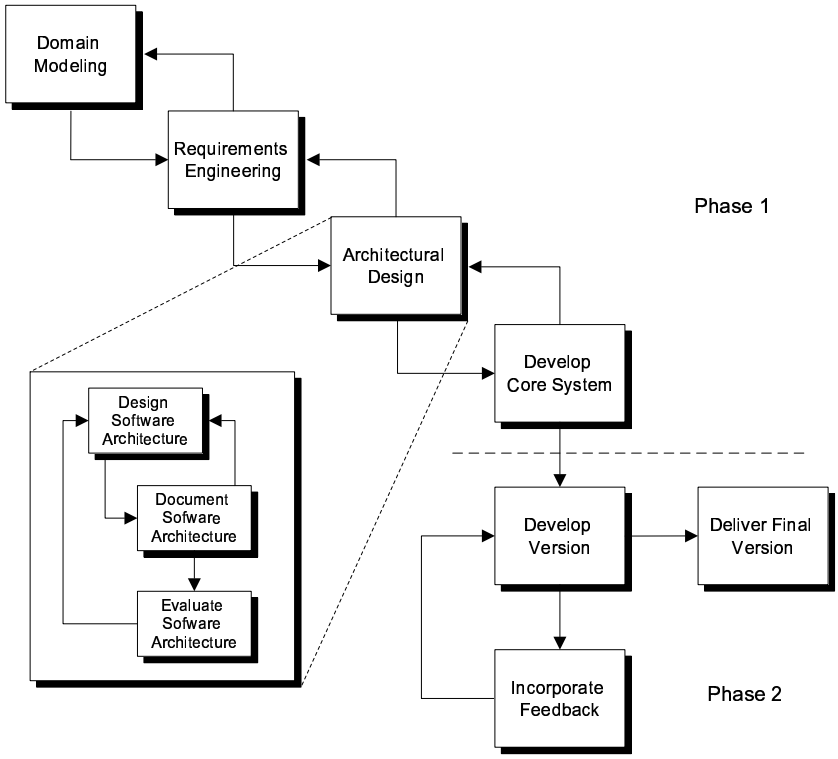
Overzicht. De rest van deze samenvatting is als volgt samengesteld. In sectie 2 geven we een beknopt overzicht van architectuur-gebaseerde software ontwikkeling. Sectie 3 introduceert het model voor gesitueerde multiagent systemen dat we ontwikkeld hebben in ons onderzoek. In sectie 4 geven we een overzicht van de referentiearchitectuur voor gesitueerde multiagent systemen. Sectie 5 bespreekt hoe we gesitueerde multiagent systemen hebben toegepast in een AGV transportsysteem. In sectie 6 tenslotte vatten we de belangrijkste bijdragen van ons onderzoek samen.

2 Architectuur-gebaseerde aanpak voor software ontwikkeling

Deze sectie geeft een korte beschrijving van een cyclus voor architectuur-gebaseerde software ontwikkeling die gebaseerd is op [16, 3]. De aanpak biedt een context waarin multiagent systemen verder gebruikt zullen worden. De ontwikkelingscyclus plaatst het ontwerp van software architectuur centraal, zie Fig. 1.

De ontwikkelingscyclus bestaat uit twee fasen: de ontwikkeling van het basissysteem en de afwerking van het finale software product. De ontwikkeling van het basissysteem is een iteratief proces waarbij het ontwerp van de architectuur (Architectural Design) itereert met vereistenanalyse (Requirements Engineering) en de ontwikkeling van het basissysteem (Develop Core System).

De ontwikkeling van de architectuur bestaat uit drie subfasen: het ontwerp van de software architectuur (Design Software Architecture), de documentatie van de architectuur (Document Software Architecture), en de evaluatie van de architectuur (Evaluate Software Architecture). Het ontwerp van de software architectuur is gebaseerd op architecturale beslissingen. Tijdens het ontwerp past de architect gekende tactieken en patronen toe om de belangrijste kwaliteitsvereisten te realiseren. Een referentiearchitectuur bestaat uit een aantal geïntegreerde patronen die hun nut bewezen hebben voor een familie van applicaties. Deze patronen kunnen een architect helpen tijdens de ontwikkeling van een software architectuur voor een systeem met gelijkaardige kenmerken en vereisten als de systemen waarvan de referentiearchitectuur is afgeleid. In ons onderzoek hebben wij gebruik gemaakt van



Figuur 1: Architectuur-gebaseerde software ontwikkeling

de Attribute Driven Design methode [7, 3] (ADD) als basis voor het ontwikkelen van software architectuur met een referentiearchitectuur.

Een software architectuur wordt beschreven aan de hand van views [12]. In ons onderzoek maken wij gebruik van verschillende viewtypes [8] voor de documentatie van software architectuur, inclusief een module viewtype (beschrijft implementatie eenheden), een component-en-connector viewtype (beschrijft runtime eenheden), en een allocatie viewtype (beschrijft de relatie tussen software elementen en de ontwikkelings- en uitvoeringsomgeving).

Een software architectuur vormt de basis van een software systeem en bepaalt daarom in belangrijke mate de kwaliteit van het systeem. Een tijdige evaluatie van de software architectuur vermijdt moeilijkheden achteraf. In ons onderzoek hebben wij de Architecture Tradeoff Analysis methode [9] (ATAM) toegepast voor de evaluatie van software architectuur.

3 Model voor gesitueerde multiagent systemen

Gesitueerde multiagent systemen zijn gebaseerd op de principes van reactieve agenten die ontwikkeld zijn in het midden van de jaren tachtig. Reactieve agenten zijn ontwikkeld als reactie op de beperkingen van cognitieve agenten. Redeneren over kennis en het plannen van acties is tijdrovend met als gevolg dat cognitieve agenten niet kunnen reageren op snelle veranderingen in de omgeving. Reactieve agenten koppelen waarneming rechtstreeks aan actie. Dit resulteert in efficiënte beslissingsmechanismen en snelle reactie in dynamische omgevingen.

Historisch kunnen een aantal families van gesitueerde agentsystemen onderscheiden worden. In de eerste generatie werden enkel systemen beschouwd met één agent. De aandacht hierbij was vooral gericht op de beslissingsarchitectuur van de agenten. Baanbrekende voorbeelden zijn de subsumption architectuur [5] en de agent netwerk architectuur [13] (ANA). In de tweede generatie werden systemen beschouwd met meerdere agenten. Een belangrijke familie zijn multiagent sys-temen die gebaseerd zijn op stigmergy waarbij agenten met mekaar communiceren door het manipuleren van de omgeving, belangrijke voorbeelden zijn synthetic ecosystems [6, 17] en computational fields [15] (Co–Fields). Een tweede familie zijn gesitueerde multiagent systemen waarbij de nadruk gelegd werd op de architectuur van de multiagent systemen, voorbeelden zijn multilayered multiagent situated systems [2] (MMASS) en block-like representation of interactive components [10] (BRIC).

Ondanks het feit dat gesitueerde agent systemen met succes zijn toegepast in de praktijk, blijven een aantal zaken open voor verder onderzoek. Enkele belangrijke uitdagingen zijn:

- Architecturen voor gesitueerde agenten leggen de nadruk op het actie-selectie mechanisme. Andere aspecten zoals waarneming en communicatie dienen te worden geïntegreerd in de agentarchitectuur.
- Gesitueerde agenten werken samen door indirecte interactie via de omgeving. Om rechtstreekse samenwerking mogelijk te maken dienen gesitueerde agenten uitgebreid te worden met sociale vaardigheden.
- De omgeving waarin de agenten zijn gesitueerd wordt typisch beschouwd als herbruikbare infrastructuur. Dit beperkt de rol van de omgeving in multiagent systemen. Om het potentieel van de omgeving tot zijn recht te laten dient de omgeving beschouwd te worden als een expliciete bouwsteen die creatief kan worden aangewend worden in het ontwerp van een multiagent systeem.

Ons onderzoek sluit aan bij vooraanstaand onderzoek in het domein en draagt bij tot de realisatie van de bovengenoemde uitdagingen. In de volgende subsecties geven we een overzicht van een geavanceerd model voor gesitueerde multiagent systemen dat we ontwikkeld hebben tijdens ons onderzoek.

3.1 De omgeving als een expliciete bouwsteen in multiagent systemen

Traditioneel kunnen drie verschillende niveaus van ondersteuning onderscheiden worden die door de omgeving aangeboden worden:

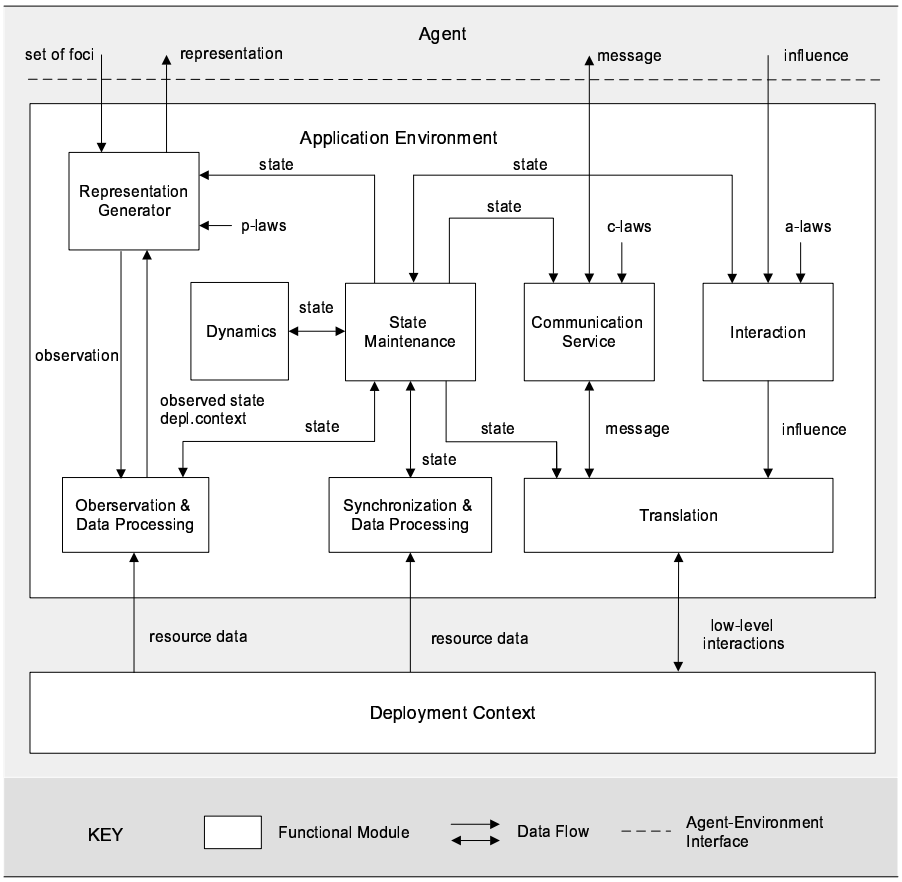
- Het *basisniveau* geeft de agenten toegang tot hulpbronnen die extern zijn aan het multiagent systeem.
- Het *abstractieniveau* zorgt ervoor dat de laagniveau details van externe hulpbronnen worden afgeschermd voor de agenten.
- Het *interactie-bemiddelingsniveau* laat toe dat de omgeving de toegang tot hulpmiddelen en de interactie tussen agenten regelt.

De verschillende types van ondersteuning door de omgeving worden typisch aangeboden in de vorm van herbruikbare infrastructuur. Een dergelijke infrastructuur biedt ondersteuning voor een specifiek soort van coördinatie voor de agenten (voorbeelden zijn digitale feromonen [6] en gradiëntvelden [14]). Andere functionaliteiten van de omgeving worden meestal niet ondersteund. De gevolgen zijn een beperkte flexibiliteit en het gebruik van ad-hoc oplossingen voor aspecten van de omgeving die niet door de infrastructuur ondersteund worden.

Wij beschouwen de omgeving als een expliciete bouwsteen in multiagent systemen. Fig. 2 geeft een overzicht van het model voor de omgeving. Het model is opgebouwd uit twee basismodules: de externe hulpbronnen (deployment context) en de toepassingsomgeving (application environment). De externe hulpbronnen zijn gegeven voor een bepaald probleem. De toepassingsomgeving daarentegen is het deel van de omgeving dat moet ontworpen worden voor een toepassing. We bespreken bondig de verschillende submodules van de toepassingsomgeving.

De **toestandsonderhoud module** (State Maintenance) vervult een centrale rol in de toepas-singsomgeving. Deze module biedt andere modules toegang tot de toestand van de toepassingsomgeving. Deze toestand bestaat typisch uit een representatie van externe hulpmiddelen (vb. de topologie van een netwerk) samen met aanvullende toestand (vb. digitale feromonen gelokaliseerd in de knopen van een netwerk).

De **representatiegenerator module** (Representation Generator) maakt selectieve waarneming van de omgeving mogelijk. Om de omgeving selectief te observeren gebruikt een agent een set van foci. Een focus bepaalt in welk type informatie de agent geïnteresseerd is. Waarneming is onderworpen aan perceptiewetten (p-laws). Perceptiewetten leggen restricties op aan wat een agent kan waarnemen gegeven een set van foci. Bijvoorbeeld, om redenen van efficiëntie kan een ontwerper een perceptiewet introduceren die restricties oplegt aan het gebied waarin een agent kan waarnemen.



Figuur 2: Model voor de omgeving

De **observatie en data verwerkingsmodule** (Observation & Data Processing) biedt de functionaliteit aan om externe hulpbronnen te observeren. De informatie afkomstig van de observatie wordt typisch verwerkt alvorens ze naar de representatiegenerator wordt doorgegeven. Een voorbeeld van een dergelijke verwerking is de integratie van sensordata.

De **interactie module** (Interaction) is verantwoordelijk voor de verwerking van acties uitgevoerd door de agenten. In ons onderzoek gebruiken we het influence–reaction model voor acties geïntroduceerd door Ferber [11]. Dit model maakt een onderscheid tussen influences en reacties. Influences worden door agenten uitgevoerd en beogen een bepaalde toestandsverandering in de omgeving. Reacties bepalen de effecten van de influences en zijn onder controle van de omgeving. In-

fluences zijn onderhevig aan actiewetten (a-laws) die restricties opleggen aan de activiteiten van agenten. Agenten kunnen influences uitvoeren die een verandering beogen in de toestand van de toepassingsomgeving (bijvoorbeeld het droppen van een digitaal feromoon) of influences die een verandering beogen in de toestand van externe hulpbronnen. Het eerste type van influence wordt uitgevoerd door de interactie module, het tweede type wordt doorgegeven aan de **vertaler** module (Translation) die de influence omzet in laagniveau acties met de externe hulpbronnen.

De **communicatiedienst module** (Communication Service) verzorgt de uitwisseling van berichten tussen agenten. Communicatie is onderworpen aan communicatiewetten (c-laws) die restricties opleggen aan de uitwisseling van boodschappen tussen agenten. Een voorbeeld is een wet slechts berichten aflevert aan agenten die zich binnen een bepaalde afstand van de zender bevinden. De communicatiedienst geeft de berichten van agenten door aan de vertaler module die de berichten omzet in laagniveau primitieven voor transmissie via een communicatie infrastructuur.

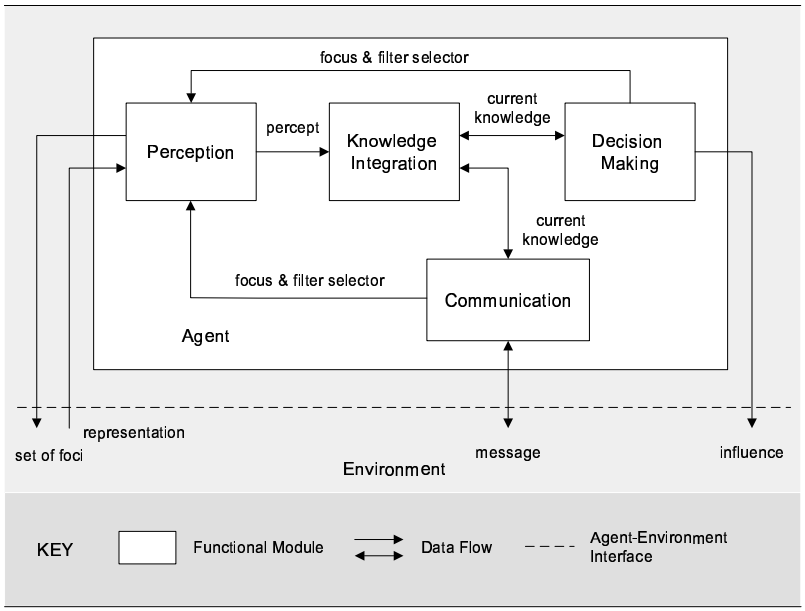
De **synchronisatie en data verwerkingsmodule** (Synchronization & Data Processing) bewaakt welbepaalde externe hulpbronnen en zorgt ervoor dat de representatie van deze hulpbronnen in de toestand van de toepassingsomgeving onderhouden wordt. Een voorbeeld is het onderhoud van de representatie van een dynamische netwerk topologie in de toestand van de toepassingomgeving.

De **dynamiek module** (Dynamics) onderhoudt processen die onafhankelijk zijn van agenten of externe hulpbronnen. Een typisch voorbeeld is de evaporatie van digitale feromonen.

3.2 Model van een gesitueerde agent

Architecturen voor gesitueerde agenten leggen de nadruk op mechanismen voor actie-selectie. Aspecten zoals waarneming en communicatie worden impliciet geïntegreerd of helemaal niet beschouwd. Wij hebben een model ontwikkeld voor gesitueerde agenten waarin deze aspecten een expliciet onderdeel uitmaken van de agentarchitectuur. Fig. 3 geeft een hoogniveau overzicht van het model voor een gesitueerde agent. De modules stellen de basis functionaliteiten voor van een agent. We geven een kort overzicht van de verschillende modules.

De **kennisintegratie module** (Knowledge Integration) omvat de kennis van de agent en biedt andere modules toegang tot deze kennis. De kennis van een agent bestaat uit twee delen. Gedeelde kennis verwijst naar een toestand die gedeeld wordt door agenten; voorbeelden zijn representaties van elementen in de omgeving en verbintenissen tussen agenten over tijdelijke samenwerkingen. Interne kennis heeft betrekking op toestand die niet gedeeld wordt tussen agenten. Een voorbeeld zijn interne parameters van een actie–selectie mechanisme.



Figuur 3: Agent model

De **waarnemingsmodule** (Perception) omvat de functionaliteit voor agenten voor selectieve waarneming. Selectieve waarneming is gebaseerd op de selectie van foci en filters. Foci worden gebruikt om welbepaalde informatie uit de omgeving waar te nemen. Filters worden gebruikt om uit de waarneming specifieke informatie te filteren. De communicatiemodule en de module voor het nemen van beslissingen gebruiken de waarnemingsmodule om de kennis van de agent in overeenstemming te brengen met de actuele toestand in de omgeving.

De **beslissingsmodule** (Decision Making). Deze module selecteert influences en voert deze uit in de omgeving. Een gesitueerde agent gebruikt een gedragsgebaseerd (behavior-based) actie-selectie mechanisme voor het nemen van beslissingen. Wij hebben gedragsgebaseerde actie-selectie mechanismen uitgebreid met de noties *rol* en *gesitueerde verbintenis*. Een rol vertegenwoordigt een coherent deel van het gedrag van een agent dat betekenis heeft in de context van een organisatie. Een gesitueerde verbintenis is een engagement van een agent om een welbepaalde rol te spelen. Het gesitueerd zijn van de verbintenis heeft betrekking op het feit dat een dergelijke verbintenis typisch geassocieerd is met factoren in de omgeving van de betrokken agent. Een gesitueerde verbintenis kan zowel verwijzen naar een engagement in een samenwerking met een andere agent, als naar een verbintenis ten opzichte van het gedrag van de agent zelf. Een voorbeeld van dit laatste is een

verbintenis van een agent om een vitale handeling te verrichten, zoals het zorgen voor de nodige energievoorraad.

De **communicatie module** (Communication) biedt de functionaliteit aan voor het uitwis-selen van berichten met andere agenten. Communicatie verloopt via een gemeenschappelijke communicatietaal. Wij hebben een model voor communicatie ontwikkeld voor gesitueerde agenten dat gebaseerd is op communicatieprotocollen. Elk protocol beschrijft een set van protocolstappen. Een protocolstap beschrijft hoe de agent zich zal gedragen in de communcatieve interactie. Communicatie is de basis voor expliciete samenwerking tussen agenten.

4 Referentiearchitectuur voor gesitueerde multi-agent systemen

Een referentiearchitectuur belichaamt de kennis en ervaring die is opgebouwd uit de ontwikkeling van een aantal toepassingen met gelijkaardige kenmerken en vereisten. Een referentiearchitectuur integreert een set van architecturale patronen die hergebruikt kunnen worden bij de ontwikkeling van nieuwe soortgelijke systemen [18, 3].

Wij hebben in ons onderzoek een referentiearchitectuur ontwikkeld voor gesitueerde multiagent systemen. Deze referentiearchitectuur is gebaseerd op de ontwikkeling van verschillende toepassingen, waaronder de pakjeswereld, een toepassing voor het delen van bestanden in een netwerk, enkele eenvoudige robotapplicaties, en een industrieel controle systeem voor automatisch bestuurde voertuigen. Tijdens het ontwerp van deze toepassingen hebben we de verschillende mechanismen ontwikkeld voor gesitueerde multiagent systemen die we besproken hebben in vorige sectie. De referentiearchitectuur integreert de verschillende functionaliteiten van agenten en de omgeving en beeldt deze functionaliteiten af op software elementen en relaties tussen de elementen. De software elementen vormen samen een abstracte systeemdecompositie die gebruikt kan worden om nieuwe soortgelijke systemen te ontwerpen.

De documentatie van de referentiearchitectuur bestaat uit een aantal views die de architectuur beschrijven vanuit verschillende standpunten. De module-decompositie view toont hoe een gesitueerd multiagent systeem is opgebouwd uit coherente implementatie eenheden. De gedeelde-data view beschrijft hoe het multi-agent systeem is gestructureerd als een set van componenten die toegang hebben tot datacomponenten. De samenwerkende-componenten view toont hoe componenten samenwerken om bepaalde systeemfunctionaliteit te realiseren. Tenslotte, de communicerende-processen view toont het multiagent systeem als een set van parallel werkende processen en hun interacties. Elke view bestaat uit een aantal viewpakketten. Een viewpakket beschrijft een welbepaald deel van de architectuur. Naast de beschrijving van de elementen en hun onderlinge relaties voorziet

elk viewpakket een aantal variatie mechanismen en een motivatie voor het ontwerp. Variatie mechanismen geven aan hoe de abstracte structuren van de referentiearchitectuur kunnen toegepast worden om een concrete software architectuur te ontwikkelen.

Ter illustratie bespreken we een viewpakket uit de communicerende-processen view. Dit viewpakket toont de belangrijkste processen en datacomponenten die betrokken zijn in waarneming, interactie, en communicatie.

4.1 Basisvoorstelling communicerende processen

Fig. 4 toont de basisvoorstelling van de communicerende processen voor waarneming, interactie, en communicatie.

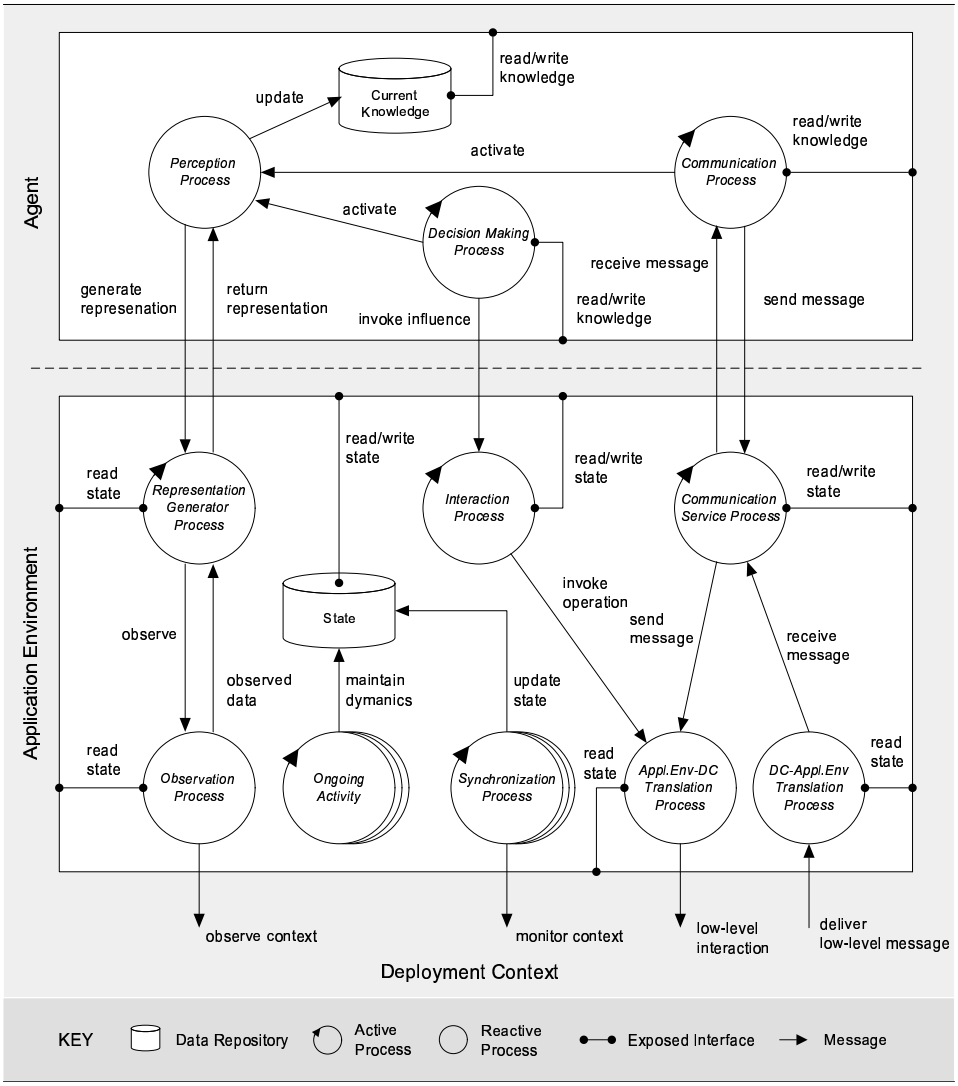
4.2 Bespreking van de elementen

Dit viewpakket toont de belangrijkste processen en datacomponenten van een gesitueerde agent en de toepassingsomgeving. We maken een onderscheid tussen *actieve* processen (Active Process) die autonoom uitvoeren en *reactieve* processen (Reactive Process) die geactiveerd worden door andere processen om een taak uit te voeren.

De bespreking van de elementen is opgedeeld in vier delen. Achtereenvolgens beschouwen we de communicerende processen voor waarneming, interactie, communicatie, en de onafhankelijke processen van de toepassingsomgeving.

Waarneming. Het `Perception Process` van de agent is een reactief proces dat kan geactiveerd worden door het `Decision Making Process` en `Communication Process`. Van zodra het perception process een aanvraag ontvangt, vraagt het aan het `Representation Generator Process` om een waarneming uit te voeren. Het representation generator process verzamelt de gevraagde toestand. Enerzijds wordt de toestand van de `State` component opgehaald en indien nodig wordt het `Observation Process` gevraagd om informatie van externe hulpbronnen te verzamelen. Het verzamelen van de toestand is onderworpen aan wetten die mogelijk beperkingen opleggen aan de waarneming. Zodra de informatie van externe hulpbronnen beschikbaar is geeft het observation process deze door aan het representation generator process. Vervolgens integreert het representation generator proces de toestand van de toepassingsomgeving met de informatie van externe hulpbronnen en geeft het resultaat (representation) terug aan het perception process van de agent. Het perception process zet de ontvangen representatie om in het juiste formaat om de toestand van de agent bij te werken in de `Current Knowledge` component. Tenslotte kan het process dat de waarneming had aangevraagd de aangepaste kennis van de agent raadplegen.

Interactie. Het `Decision Making Process` is een actief proces van de agent dat influences selecteert en uitvoert in de omgeving. Het `Interaction Process`



Figuur 4: Communicerende processen voor waarneming, interactie, en communicatie

verzamelt de influences van de agenten en zet deze om in operaties (operations). De uitvoering van operaties is onderworpen aan actiewetten die beperkingen opleggen aan de activeiten van agenten in de omgeving. Operaties die een aanpassing van de toestand van de omgeving beogen worden rechtstreeks uitgevoerd door het

interaction process. Operaties die een aanpassing van de toestand van externe hulpbronnen beogen worden doorgestuurd naar het `Appl.Env-DC Translation Process`. Dit proces zet de operaties om in laagniveau acties en voert die uit op de externe hulpbronnen.

Communicatie. Het `Communication Process` is een actief proces dat verantwoordelijk is voor de uitwisseling van boodschappen met andere agenten. Een nieuw bericht wordt doorgegeven aan het `Communication Service Process` dat de communicatiewetten toepast en vervolgens het bericht doorgeeft aan het `Appl.Env-DC Translation Process`. Dit proces zet het bericht om in een laagniveau boodschap die verstuurd kan worden via de externe communicatie infrastructuur. Het `DC-Appl.Env Translation Process` verzamelt inkomende boodschappen en zet deze om in een gepast formaat voor de agenten. De boodschappen worden doorgegeven aan het communication service process dat de berichten aflevert aan de gepaste agenten.

Onafhankelijke processen in de toepassingsomgeving. `Synchronization Processes` zijn actieve processen die de toestand van applicatie-specifieke hulpbronnen observeren en de representatie ervan in de toestand van de toepassingsomgeving in overeenstemming houden. `Ongoing Activities` zijn actieve processen die dynamiek in de applicatieomgeving representeren die onafhankelijk is van agenten en externe hulpbronnen. Deze processen passen voortdurend de toestand van de toepassingsomgeving aan overeenkomstig de vereisten van de applicatie.

4.3 Motivatie voor het ontwerp

Agenten beschikken over twee actieve processen, één voor het selecteren van influences en één voor communicatie. Deze aanpak laat toe dat beide processen in parallel uitvoeren wat de efficiëntie van het nemen van beslissingen ten goede komt. De communicatie tussen de twee processen gebeurt indirect via current knowledge wat zorgt voor een goede ontkoppeling. Het perception process is reactief, een agent zal enkel een aanvraag tot waarneming uitvoeren wanneer dit vereist is voor het nemen van beslissingen.

De toepassingsomgeving beschikt over actieve processen voor het afhandelen van waarneming, interactie, en communicatie. Het observation process is reactief, dit proces verzamelt enkel informatie van externe hulpbronnen wanneer nodig. De translation processen zijn eveneens reactief, deze processen bieden hun diensten aan op vraag van andere processen. De synchronization processen en de ongoing activities tenslotte zijn actieve processen die hun taken volbrengen onafhankelijk van de rest van de activiteit in het systeem.

Doordat actieve processen in parallel kunnen uitvoeren wordt het mogelijk verschillende opdrachten in het systeem tegelijk af te werken. Reactieve processen daarentegen worden enkel geactiveerd wanneer nodig. Deze aanpak zorgt ervoor

dat de hulpbronnen in het systeem op een efficiënte manier gebruikt worden.

5 AGV transportsysteem

Een automatisch bestuurd voertuig (Automatic Guided Vehicle, AGV) is een volledig geautomatiseerd voertuig dat transportopdrachten kan uitvoeren in een industriële omgeving. AGV transportsystemen met meerdere AGV's worden gekenmerkt door voortdurende dynamiek. De stroom van taken die het systeem moet verwerken is onregelmatig, AGV's kunnen het systeem tijdelijk verlaten bijvoorbeeld voor onderhoud, productiemachines hebben variable wachttijden, e.d. Traditioneel worden AGV transportsystemen bestuurd door een centraal controle systeem. De belangrijkste voordelen van deze aanpak zijn de beschikbaarheid van een centraal configuratiepunt en de voorspelbaarheid van het gedrag van het systeem.

In een project in samenwerking met Egemin hebben wij een inovatieve versie ontwikkeld voor de AGV controle software [1]. In dit project hebben we een gedecentraliseerd controle systeem ontwikkeld, gebaseerd op een gesitueerd multiagent systeem. De bedoeling was te onderzoeken in hoeverre een decentraliseerde architectuur kan bijdragen tot het verhogen van de flexibiliteit en de openheid van het systeem.

De succesvolle ontwikkeling van deze industriële applicatie heeft in belangrijke mate bijgedragen tot de ontwikkeling van de referentiearchitectuur voor gesitueerde multiagent systemen. In deze sectie geven we een hoogniveau overzicht van het gesitueerde multiagent systeem voor het AGV transportsysteem.

5.1 Belangrijkste systeemvereisten

De belangrijkste functionaliteit van het systeem is het afhandelen van transporten, d.w.z. ladingen (zoals paletten) van de ene naar een andere plaats brengen. Het afhandelen van een transport houdt de volgende deeltaken in:

1. Transporttoewijzing: transporten worden gegenereerd door klantsystemen (typisch een warehouse management systeem) en dienen te worden toegewezen aan AGV's.

2. Routering: AGV's dienen een efficiënte weg te vinden op de layout van de fabrieksvloer, AGV's mogen enkel langs voorgedefinieerde paden manoeuvreren.

3. Verzamelen van verkeersinformatie: om efficiënt naar een bestemming te rijden dienen de AGV's rekening te houden met de wijzigende verkeerstoestand in het systeem.

4. Vermijden van botsingen: vanzelfsprekend mogen AGV's niet op hetzelfde ogenblik een kruispunt oversteken; doch botsingen moeten ook vermeden

worden wanneer twee AGV's mekaar passeren in dicht bij elkaar gelegen parallelle wegen.

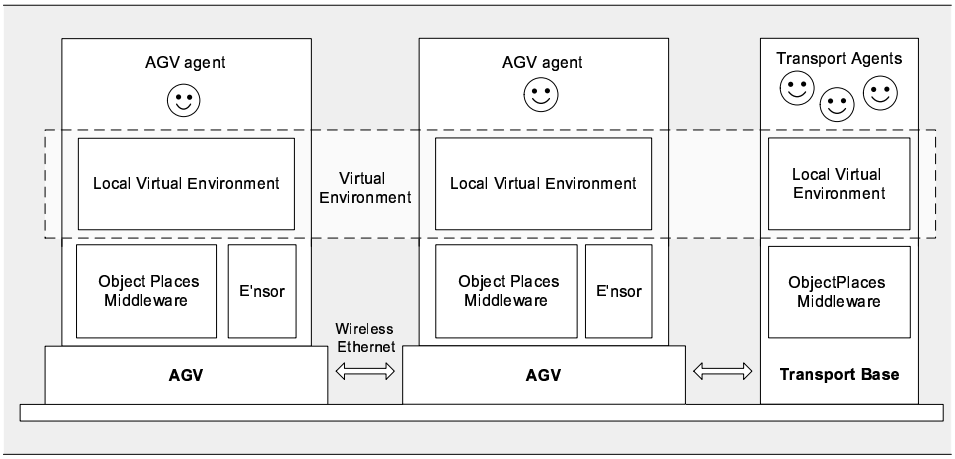
5. Vermijden van deadlocks: AGV's moeten ervoor zorgen dat er geen deadlock situatie kan ontstaan.

Naast de functionele vereisten zijn er ook kwaliteitsvereisten. Belangrijke traditionele kwaliteitsvereisten zijn performantie, configureerbaarheid, en robustheid. Daarnaast winnen flexibiliteit en openheid steeds meer aan belang. Flexibiliteit laat toe dat het systeem opportuniteiten uitbuit, mogelijke problemen anticipeert, e.d. Openheid zorgt ervoor dat het transportsysteem zelfstandig kan omgaan met AGV's die aan het systeem worden toegevoegd of die het systeem verlaten.

Het doel van het project met Egemin was te onderzoeken in hoeverre een gedecentraliseerde architectuur gebaseerd op een gesitueerd multiagent systeem in staat is te voldoen aan deze systeemvereisten.

5.2 Gesitueerd multiagent systeem

Fig. 5 geeft een hoogniveau model van het AGV transportation system.



Figuur 5: Hoogniveau model van het AGV transportsysteem

Agenten. We hebben twee types agenten geïntroduceerd: AGV agenten en transport agenten. De keuze om elke AGV te controleren door een AGV agent is vanzelfsprekend. Doordat transporten in onderhandeling dienen toegekend te worden aan de meest geschikte AGV hebben we transport agenten geïntroduceerd.

Een AGV agent is verantwoordelijk voor de controle van het AGV voertuig waarmee het geassocieerd is. Aldus wordt een AGV een autonome entiteit die kan

inspelen op opportuniteiten die zich voordoen in de omgeving, en die het systeem kan verlaten en opnieuw betreden zonder de rest van het systeem te verstoren.

Een transport agent vertegenwoordigt een transport in het systeem. Deze agent is verantwoordelijk voor de toewijzing van het transport aan een AGV agent, en de interactie met de klant die de opdracht heeft gegeven. Transport agenten bevinden zich op een transportbasis (transport base). Een transportbasis is een stationaire computer die zich bevindt in de fabrieksruimte.

Beide types agenten hebben een gelijkaardige architectuur die overeenkomt met de agentarchitectuur gedefinieerd in de referentiearchitectuur. De interne structuur van de twee types agenten is echter verschillend overeenkomstig de taken die beide types agenten dienen te vervullen.

Het toepassen van een gesitueerd multiagent systeem draagt als volgt bij tot flexibiliteit en openheid van het systeem: (1) gesitueerde agenten handelen lokaal, dit zorgt ervoor dat agenten hun gedrag beter kunnen aanpassen aan de veranderingen in de omgeving en dus beter kunnen inspelen op opportuniteiten—dit is een belangrijk kenmerk voor flexibiliteit; (2) gesitueerde agenten zijn autonome entiteiten die samenwerken met andere agenten in hun directe nabijheid; agenten kunnen mekaars omgeving naar goeddunken betreden en verlaten—dit is een belangrijk kenmerk voor openheid.

Virtuele omgeving. (Virtual Environment) Om hun taken te vervullen dienen AGV agenten en transport agent hun activiteiten te coördineren. Coördinatie is nodig voor het toekennen van taken, het vermijden van botsingen, e.d. Wij hebben voor een aanpak gekozen waarbij agenten hun activiteiten coördineren via de omgeving.

AGV's zijn gesitueerd in de fysieke wereld. De omgeving van AGV's biedt echter weinig mogelijkheden voor coördinatie. AGV's kunnen slechts voortbewegen over de voorgedefinieerde paden, ze kunnen ladingen oppikken en afzetten, en ze kunnen communiceren via een draadloos netwerk. Om de mogelijkheden voor coördinatie uit te breiden hebben we een virtuele omgeving (virtual environment) geïntroduceerd. Deze virtuele omgeving biedt een medium voor de AGV agenten en transport agenten om informatie uit te wisselen en hun gedrag te coördineren. Bovendien schermt de virtuele omgeving laagniveau details af voor de agenten, zoals communicatie over het draadloos netwerk en de fysieke besturing van de AGV voertuigen.

Doordat AGV agenten en transport agenten gedistribueerd zijn over verschillende machines is de virtuele omgeving eveneens gedistribueerd over de AGV's en de transportbasis. Dit impliceert dat elke AGV en de transportbasis een lokale virtuele omgeving (local virtual environment) voorzien. De toestand van deze lokale virtuele omgevingen wordt consistent gehouden voor zover nodig. In het AGV transportsysteem is er geen software entiteit die overeenkomt met de toepassingsomgeving zoals die is gedefinieerd in de referentiearchitectuur voor gesitueerde multiagent systemen. In plaats daarvan zijn instanties van de lokale virtuele

omgeving voorzien op elke machine in het AGV transportsysteem. De instanties van de lokale virtuele omgeving verschillen voor de AGV's en de transportbasis. Bijvoorbeeld, de lokale virtuele omgeving op AGV's voorziet in een hoogniveau interface die AGV agenten toelaat de AGV machine te besturen. Dergelijke functionaliteit wordt vanzelfsprekend niet aangeboden door de lokale virtuele omgeving op de transportbasis.

Voor de synchronisatie van de toestand tussen naburige lokale virtuele omgevingen wordt gebruik gemaakt van de ObjectPlaces middleware [21, 22, 23]. ObjectPlaces ondersteunt de coördinatie van knopen in een mobiel netwerk. De laagniveau controle van de AGV voertuigen wordt verzorgd door E'nsor[1]. We hebben de E'nsor software volledig hergebruikt in het project. E'nsor is uitgerust met een kaart van de omgeving die de paden waarover AGV's kunnen rijden verdeelt in segmenten en knopen. E'nsor biedt een interface aan om de AGV per segment te sturen. Voorbeelden van instructies zijn: `Move(segment)` wat de AGV stuurt over het gegeven segment, en `Pick(segment)` wat de AGV stuurt over het gegeven segment en vervolgens de AGV de opdracht geeft de lading daar op te nemen. De laagniveau controle van deze instructies worden door E'nsor afgehandeld. E'nsor biedt bovendien een interface aan waarmee de toestand van de machine kan geïnspecteerd worden; voorbeelden zijn de positie van de AGV en de status van de batterij.

Coördinatie via de virtuele omgeving. We illustreren met enkele voorbeelden hoe de agenten in het AGV transportsysteem de virtuele omgeving gebruiken voor coördinatie.

Routering. De lokale virtuele omgeving heeft een kaart met de paden waarover AGV's kunnen rijden. Deze kaart is uitgerust met informatie omtrent de afstand tussen verschillende locaties in het systeem, vergelijkbaar met wegwijzers in het verkeer. AGV agenten kunnen deze informatie gebruiken om het kortste pad naar een bepaalde bestemming te vinden.

Transport toekenning. Doordat transporten op willekeurige tijdstippen ontstaan is het efficiënt toekennen van transporten aan AGV's complex. Om met deze dynamiek om te gaan hebben we een veld-gebaseerde aanpak ontwikkeld voor het toekennen van transporten aan AGV's. In deze aanpak zenden transport agenten velden uit in de virtuele omgeving die vrije AGV's aantrekken. Om te vermijden dat meerdere AGV's naar hetzelfde transport rijden, sturen de AGV agenten velden uit die andere AGV's afstoten. Elke AGV die op zoek is naar een transport combineert de velden die het ontvangt en volgt vervolgens de gradiënt van dit gecombineerde veld hetwelk de AGV leidt naar een lading. Doordat AGV agenten voortdurend de situatie herbekijken en de taaktoekenning maar definitief is bij het opnemen van de lading onstaat een uiterst flexibele oplossing.

[1] E'nsor® is een acroniem voor Egemin Navigation System On Robot.

Vermijden van botsingen. AGV agenten vermijden botsingen via coördinatie van hun bewegingen in de virtule omgeving. AGV agenten reserveren het pad waarover ze rijden in de lokale virtuele omgeving met omhullenden (hulls). Een omhullende markeert het fysische gebied dat een AGV in beslag neemt. Een reeks van omhullenden beschrijft het fysische gebied dat een AGV in beslag zal nemen tijdens een beweging over een pad. Wanneer een gebied in de virtuele omgeving slechts gemarkeerd is door één AGV dan kan deze meteen doorrijden. Doch wanneer de omhullenden van meerdere AGV's overlappen dan geeft de lokale virtuele omgeving voorrang aan de AGV met de hoogste prioriteit (er is een volledige ordening op basis van transporten en AGV's). Nadat een AGV een gereserveerd gebied is gepasseerd zal het de markering in de omgeving verwijderen.

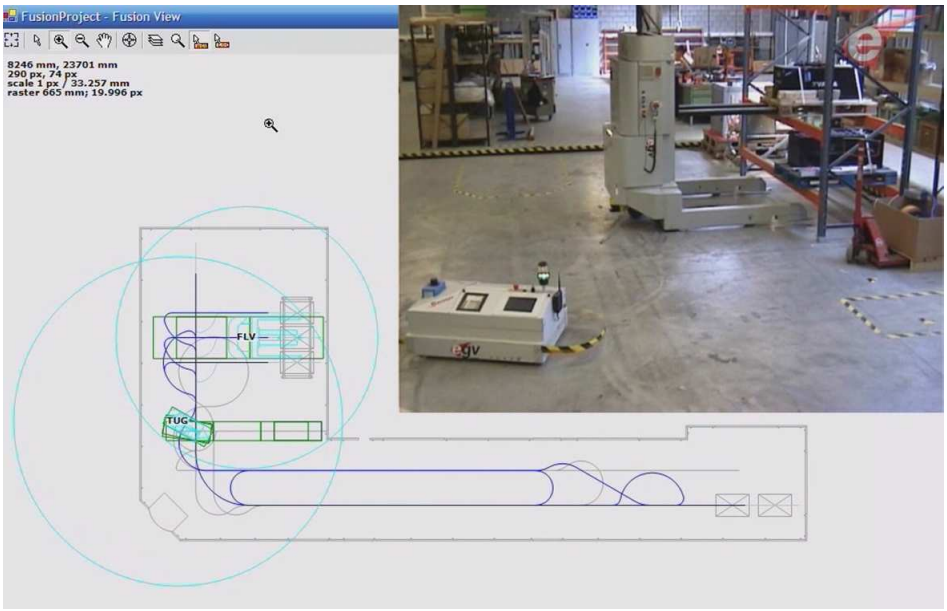
Deze voorbeelden tonen aan hoe de lokale virtuele omgeving bijdraagt tot een flexibele coördinatie van de agenten in het AGV transportsysteem. Agenten coördineren door het aanbrengen van markeringen in de lokale virtuele omgeving en het observeren van markeringen van andere agenten. Het afhandelen van coördinatie tussen agenten in de omgeving zorgt voor een betere scheiding van belangen en een betere beheersbaarheid van de complexiteit.

5.3 Evaluatie

Voor de evaluatie van de software architectuur van het AGV transportsysteem hebben we een ATAM uitgevoerd [26, 30, 4]. Deze evaluatie was een uiterst leerrijke ervaring. Een belangrijk besluit uit de evaluatie was de tradeoff in een gedecentraliseerd systeem (zoals het gesitueerde multiagent systeem) tussen flexibiliteit en de vereiste bandbreedte voor communicatie. In vergelijking met de bestaande gecentraliseerde aanpak biedt het gesitueerde multiagent systeem een flexibelere oplossing, doch de toemenende kost voor communicatie die hiermee gepaard gaat is significant. Testen na de ATAM hebben aangetoond dat het gebruik van de beschikbare bandbreedte binnen de gestelde beperkingen blijft. Als een proof-of-concept hebben we een demonstrator ontwikkeld met twee AGV's die de basisfunctionaliteit ondersteunt voor routering, het afhandelen van transporten, het vermijden van botsingen, en het vermijden van deadlock. Fig. 6 toont een snapshot van de AGV's in actie, tesamen met een samengesteld zicht (fusion view) op de virtuele omgeving.

6 Conclusies

We sluiten deze samenvatting af met een overzicht van de bijdragen van ons onderzoek. Vooreerst hebben we een geavanceerd model ontwikkeld voor gesitueerde multiagent systemen dat een belangrijke uitbreiding betekent op vooraanstaande aanpakken in het domein van gesitueerde multiagent systemen. Ten tweede, uit onze ervaring met het bouwen van verschillende toepassingen hebben we een ref-



Figuur 6: Demonstrator met AGV's in actie

erentiearchitectuur ontwikkeld voor gesitueerde multiagent systemen. Tenslotte hebben we de toepasbaarheid van de architectuur-gebaseerde aanpak voor software ontwikkeling met gesitueerde multiagent systemen gevalideerd in een complexe industriële applicatie. Concrete bijdragen zijn:

- We hebben een nieuw perspectief ontwikkeld op de rol van de omgeving in multiagent systemen [34, 38, 42, 28, 33]. In het bijzonder hebben we de omgeving gepromoveerd tot een expliciete bouwsteen die creatief kan geëxploiteerd worden tijdens het ontwerp van een multiagent systeem.

- We hebben vooraanstaande aanpakken in gesitueerde multiagent systemen uitgebreid met een geavanceerd model voor gesitueerde agenten [27, 41, 39, 24, 40]. Dit model biedt ondersteuning voor selectieve waarneming, sociaal gedrag met rollen en gesitueerde verbintenissen en protocol-gebaseerde communicatie.

- Uit onze ervaring met het bouwen van applicaties met gesitueerde multiagent systemen hebben we een referentiearchitectuur ontwikkeld voor deze systemen [32, 29, 31]. Deze referentiearchitectuur ondersteunt de ontwikkeling van nieuwe software architecturen voor toepassingen met gelijkaardige kenmerken en vereisten. Om de haalbaarheid van de referentiearchitectuur

aan te tonen hebben we een raamwerk ontwikkeld dat de architectuur implementeert en we hebben dit raamwerk toegepast voor de ontwikkeling van een aantal prototype applicaties.

- We hebben een gesitueerd multiagent systeem toegepast in een complex industrieel transportsysteem voor AGV's [37, 36, 35, 23]. De inzichten die we verworven hebben uit dit project hebben in belangrijke mate bijgedragen tot de ontwikkeling van de referentiearchitectuur. Het ontwerp van de software architectuur van dit systeem, de ontwikkeling van de software en de evaluatie van de applicatie tonen aan dat gesitueerde multiagent systemen kunnen toegepast worden in complexe applicaties waar flexibiliteit en openheid belangrijke kwaliteitesvereisten zijn.

We hebben in dit onderzoek een architectuur-gebaseerde aanpak voorgesteld voor software ontwikkeling met multiagent systemen. Het toepassen van deze aanpak in een complexe industriële toepassing heeft ons ervan overtuigd dat de integratie van multiagent systemen als software architectuur in een algemeen software ontwikkelingsproces een sleutel is tot industriële aanvaarding van multiagent systemen.

Referenties

[1] EMC^2: Egemin Modular Controls Concept, Project Supported by the Institute for the Promotion of Innovation Through Science and Technology in Flanders (IWTVlaanderen), (8/2006). http://emc2.egemin.com/.

[2] S. Bandini, S. Manzoni, and C. Simone. Dealing with Space in Multiagent Systems: A Model for Situated Multiagent Systems. In *1st International Joint Conference on Autonomous Agents and Multiagent Systems*. ACM Press, 2002.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley Publishing Comp., 2003.

[4] N. Boucke, 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.

[5] R. Brooks. Achieving artificial intelligence through building robots. *AI Memo 899, MIT Lab*, 1986.

[6] S. Brueckner. *Return from the Ant, Synthetic Ecosystems for Manufacturing Control*. Ph.D Dissertation, Humboldt University, Berlin, Germany, 2000.

[7] F. Buchmann and L. Bass. Introduction to the Attribute Driven Design Method. In *23rd International Conference on Software Engineering*, Toronto, Ontario, Canada, 2001. IEEE Computer Society.

[8] 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.

[9] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison Wesley Publishing Comp., 2002.

[10] J. Ferber. *An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, 1999.

[11] 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.

[12] P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, 1995.

[13] P. Maes. Situated Agents can have Goals. *Designing Autonomous Agents, MIT Press*, 1990.

[14] M. Mamei and F. Zambonelli. Co-Fields: A Physically Inspired Approach to Distributed Motion Coordination. *IEEE Pervasive Computing*, 3(2):52–61, 2004.

[15] M. Mamei and F. Zambonelli. *Field-based Coordination for Pervasive Multi-agent Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[16] S. McConell. Rapid Development: Taming Wild Software Schedules. *Microsoft Press*, 1996.

[17] 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.

[18] P. Reed. Reference Architecture: The Best of Best Practices. *The Rational Edge*, 2002. www-128.ibm.com/developerworks/rational/library/2774.html.

[19] G. Roman, C. Julien, and J. Payton. A Formal Treatment of Context-Awareness. *7th International Conference on Fundamental Approaches to Software Engineering*, 2004.

[20] N. Rozanski and E. Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison Wesley Publishing Comp., 2005.

[21] 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.

[22] K. Schelfthout, D. Weyns, and T. Holvoet. Middleware for Protocol-based Coordination in Dynamic Networks. In *3rd International Workshop on Middleware for Pervasive and Ad-hoc Computing*, Grenoble, France, 2005. ACM Press.

[23] 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.

[24] 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.

[25] R. Want. System Challenges for Pervasive and Ubiquitous Computing (Intel). *Invited talk, International Conference on Software Engineeging, St. Louis, USA*, 2005.

[26] D. Weyns, N. Boucke, 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.

[27] D. Weyns and T. Holvoet. Formal Model for Situated Multi-Agent Systems. *Fundamenta Informaticae*, 63(1-2):125–158, 2004.

[28] D. Weyns and T. Holvoet. On Environments in Multiagent Systems. *AgentLink Newsletter*, 16:18–19, 2005.

[29] D. Weyns and T. Holvoet. A Reference Architecture for Situated Multiagent Systems. In *Environments for Multiagent Systems III, 3th International Workshop, E4MAS, Hakodate, Japan, 2006*, Lecture Notes in Computer Science. Springer, 2006.

[30] 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.

[31] D. Weyns and T. Holvoet. Multiagent systems and Software Architecture. In *Special Track on Multiagent Systems and Software Architecture, Net.ObjectDays*, Erfurt, Germany, 2006.

[32] D. Weyns and T. Holvoet. Multiagent Systems and Software Architecture: Another Perspective on Software Engineering with Multiagent Systems. In *5th International Joint Conference on Autonomous Agents and Multi-Agent Systems, AAMAS, Hakodate, Japan*, 2006.

[33] D. Weyns, A. Omicini, and J. Odell. Environment as a First-Class Abstraction in Multiagent Systems. *Autonomous Agents and Multi-Agent Systems*, 14(1), 2007.

[34] D. Weyns, H. V. D. Parunak, F. Michel, T. Holvoet, and J. Ferber. Environments for Multiagent Systems, State-of-the-art and Research Challenges. Lecture Notes in Computer Science, Vol. 3374. Springer Verlag, 2005.

[35] D. Weyns, K. Schelfthout, and T. Holvoet. Architectural design of a distributed application with autonomic quality requirements. In *ICSE Workshop on design and evolution of autonomic application software, St. Louis, Missouri*, New York, NY, USA, 2005. ACM Press.

[36] 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.

[37] D. Weyns, K. Schelfthout, T. Holvoet, T. Lefever, and J. Wielemans. Architecture-centric development of an AGV transportation system. In *Multi-Agent Systems and Applications IV, 4th International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS, Budapest, Hungary*, Lecture Notes in Computer Science, Vol. 3690. Springer, 2005.

[38] D. Weyns, M. Schumacher, A. Ricci, M. Viroli, and T. Holvoet. Environments for Multiagent Systems. *Knowledge Engineering Review*, 20(2):127–141, 2005.

[39] 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.

[40] 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.

[41] D. Weyns, E. Steegmans, and T. Holvoet. Towards Active Perception in Situated Multi-Agent Systems. *Applied Artificial Intelligence*, 18(9-10):867–883, 2004.

[42] D. Weyns, G. Vizzari, and T. Holvoet. Environments for situated multiagent systems: Beyond Infrastructure. In *Proceedings of the Second International Workshop on Environments for Multi-Agent Systems, Utrecht, 2005*, Lecture Notes in Computer Science, Vol. 3380. Springer Verlag.

[43] F. Zambonelli and H. V. D. Parunak. *From Design to Intention: Signs of a Revolution.* 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems, Bologna, Italy, ACM Press, New York, 2002.