Architectural Design of a Distributed Application with Autonomic Quality Requirements

Danny Weyns, Kurt Schelfthout and Tom Holvoet
AgentWise, DistriNet, Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{Danny.Weyns, Kurt Schelfthout, Tom.Holvoet}@cs.kuleuven.ac.be

Abstract

An autonomic system is essentially characterized by quality requirements that specify that the system should be able to adapt itself (configure, optimize, heal, etc.) under varying circumstances and situations. These quality requirements call for an architecture centric software engineering approach. In this paper, we discuss and illustrate the architectural design of a complex real-world distributed application with autonomic quality requirements. In particular, we present an architecture with autonomous entities (agents) for managing warehouse logistics. We illustrate how the subsequent architectural decisions are guided by a reference architecture for situated multi-agent systems on the one hand, and by functional and quality requirements of the application on the other hand.

1. Introduction

Software architecture is generally acknowledged as a crucial part of the design of a software system. During architectural design, the architect "shapes" the structures of the software system. The software architecture is the backbone of the designed solution, it meets the functional requirements of the system and aims to satisfy the essential quality requirements.

The software architecture of a systems comprises different structures, reflected in different *architectural views* [1]. Most common is the module view that provides modules as containers for holding responsibilities and data flow relationships among the modules. Other views are the concurrency view that focuses on dynamic aspects of the system such as parallelism and synchronization, or the deployment view that describes the allocation of the available processors in the system.

Software systems are built to perform particular functionality. However, various stakeholders interested in the software system (project managers, developers, end users, customers, maintainers, etc.) will have additional requirements regarding the *quality* of the software system such as performance, scalability, adaptability, maintainability, etc. Today, software engineers generally recognize that quality requirements of a system are primarily achieved through its software architecture. Many *architectural patterns* [13] have been developed to realize particular quality requirements. A couple of well-known patterns are client-server, pipe-and-filter, layers, or blackboard. Reference architectures [1] go one step further in reuse of best practices in architectural design. A reference architecture defines an abstract architecture that serves as a blueprint to develop software architectures for a family of applications that are characterized by specific functional and quality requirements.

Autonomic systems and quality requirements. Autonomic computing is recognized as a viable solution to challenge the increasing complexity of managing software systems. To be autonomic, a system must be able to adapt itself under varying circumstances. Examples of autonomic capabilities are self-configuration, self-optimization, or self-healing [8]. Since autonomic systems are essentially characterized by quality requirements that specify that the system should be able to adapt itself in changing situations, the design and development of autonomic systems call for an architecture centric software engineering approach. The importance of architecture for autonomic systems is emphasized by many researchers, see e.g. [8, 6, 4].

In our research, we study the engineering of complex distributed applications with autonomic quality requirements. Example domains are self-managing networks or decentralized control of logistic machines in a warehouse. Flexibility, adaptability and openness are important quality properties for systems to be able to adapt autonomously. We put forward situated multiagent systems (situated MASs) as an approach to build such complex applications. We have developed a reference architecture for situated MASs to guide the development of applications with autonomic quality requirements. In this paper, we discuss and illustrate the architectural design of a complex real-world application with autonomic properties. We show how the subsequent architectural decisions are guided by the reference architecture for situated MASs on the one hand, and by functional and quality requirements of the application on the other hand.

DEAS 2005 May 21, 2005, St. Louis, Missouri, USA.

Overview of the paper. The remainder of this paper is structured as follows. Section 2 elaborates on situated MASs and the reference architecture we have developed for situated MASs. In section 3, we discuss the architectural design of a transportation system for managing warehouse logistics, and illustrate how we have applied the reference architecture to the design of this real-word application. Finally, we draw conclusions in section 4.

2. A reference architecture for situated multiagent systems

In this section we briefly introduce situated MASs and motivate the use of situated MASs for the design of autonomic applications. Next, we give a high-level overview of the reference architecture we have developed for situated MASs.

2.1 Situated multiagent systems

A situated MAS[1] consists of a (distributed) environment populated with a set of autonomous entities (agents) that cooperate to solve a complex problem in a decentralized way. Situated agents have 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. A situated agent does not use long-term planning to decide what action sequence should be executed, but selects actions on the basis of its current position, the state of the world it perceives and limited internal state. Intelligence in a situated MAS originates from the interactions between the agents, rather than from their individual capabilities.

In situated MASs, agents and the environment are first-order abstractions [17]. Situated agents exploit the environment to share information and coordinate their actions. A digital pheromone, for example, 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 pheromone paths to locations of interest. Another example is a gradient field that propagates through the environment and changes in strength the further it is propagated. Agents can use a gradient field as a guiding beacon. Situated MASs have been applied with success in practical applications over a broad range of domains. Some examples are manufacturing control [10], supply chains systems [11], and network management [2].

Cooperating agents situated in an environment is a natural concept to manage complexity in a decentralized manner. Agents encapsulate their own behavior and are able to adapt to changes in their environment. Well known benefits of situated MASs are efficiency, robustness and flexibility [21]. These fundamental properties make situated MASs a suitable approach for building self-managing applications.

2.2 Reference architecture in a nutshell

In the last three years, we have been developing a reference architecture for situated MASs. This reference architecture embodies the knowledge and experiences we have acquired during our research. The reference architecture generalizes and extracts common functions and structures from various experimental applications we have studied, including a simple peer-to-peer file sharing system[2], a simulation of an automatic guided vehicle transportation system[3], and several basic robot applications. Currently, the reference architecture is applied in an AGV (Automatic Guided Vehicle) transportation system. We elaborate on this complex real-world application in section 3.

The goal of the reference architecture is to offer support to software engineers for the architectural design of complex decentralized applications with autonomic properties. In the first place, we focus on systems that require self-managing behavior with respect to dynamism and change. In particular, the system must be able to cope autonomously with changing circumstances: new agents may join the system, others may leave, or the environment may change, e.g. its topology, or its characteristics such as throughput and accessibility. Self-configuration and self-protection as discussed in [8] are not the first concerns we focus on. The table below lists the properties of the problem domains we focus on, the corresponding target quality properties of the reference architecture, and the basic principles we apply to realize these quality properties. Flexibility,

PROPERTIES PROBLEM DOMAIN	*TARGET QUALITY PROPERTIES ARCHITECTURE*	*PRINCIPLES*
dynamism change	flexibility adaptability openness	decentralized control collective behavior adaptive behavior
complexity	manageability	modularity loose coupling separation of concerns
	unambiguity	formalization

adaptability and openness are target quality properties for the reference architecture to cope with dynamism and change. Flexibility enables a system to cope with different environmental situations, adaptability refers to a system's capability to adapt its behavior over time with changing circumstances, and openness enables a system to cope with expansion (new agents that join the system) and reduction (agents that leave the system). The main software engineering principles we apply to realize these quality properties are decentralized control, collective behavior and adaptive behavior. Manageability and unambiguity are target quality properties of the reference architecture to cope with complexity. The main software engineering principles we apply to realize these quality properties are modularity, loose coupling, and separation of concerns; and we provide a formal specification of the reference architecture.

We now give a brief overview of the main modules of the reference architecture. Fig. 1 depicts a high-level module view of the reference architecture for situated MAS. The architecture integrates three primary abstractions: *agents*, *ongoing activities* and the *environment*. We successively look at the architecture of each abstraction.

Agents. The agent architecture models different concerns of the agent (perception, decision making and communication) as sep-

[1] Alternative descriptions are behavior-based agents [3], adaptive autonomous agents [9] or hysteretic agents [7][5].

[2] http://trappie.studentenweb.org/andy/www/site_mai/main.php

[3] http://www.cs.kuleuven.ac.be/∼distrinet/taskforces/agentwise/agvsimulator/

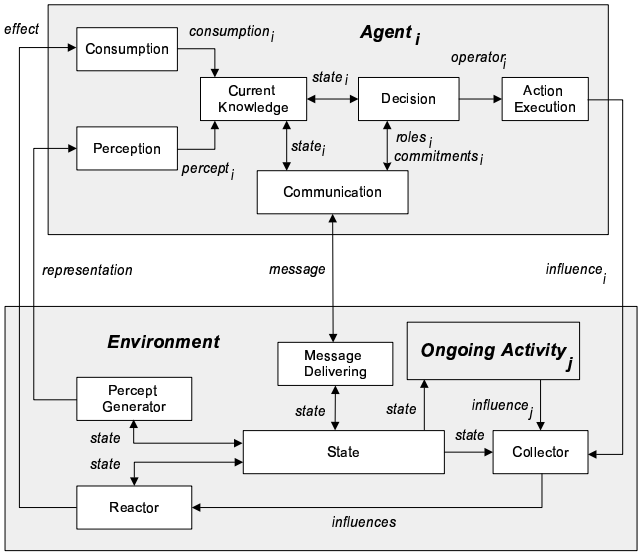

Figure 1. High-level module view of the reference architecture.

arate modules. The $Perception$ module maps the local state of the environment onto a percept for the agent. We developed a model for active perception that enables an agent to direct its perception at the most relevant aspects in the environment according to its current task [20]. The $Consumption$ module "consumes" consumptions for the agent. A consumption is an effect of the reaction of the environment for a particular agent. When an agent consumes a consumption, the consumed effect can be absorbed (e.g. the agent acquires energy), the agent may simply hold an element (e.g. an object it has picked up) or the consumption may affect the agent's state (e.g. the arm of a robot is wrenched through an external force). The $CurrentKnowledge$ module integrates the percepts and consumptions to update the current knowledge of the agent. The $Decision$ module is responsible for action selection [14]. We developed the decision module as a free-flow architecture. Free-flow architectures allow flexible and adaptive action selection [15]. Since existing free-flow architectures lack explicit support for social behavior, we introduced the concepts of a *role* and a *situated commitment*. A role covers a logical functionality of the agent, while a situated commitment allows an agent to adjust its behavior towards the role in its commitment. An agent can commit to itself, e.g. when it has to fulfill a vital task. However, in a collaboration agents commit relatively to one another via communication. Roles and situated commitments are building blocks for collective behavior. The operator selected by the decision module is passed to the $ActionExecution$ module that invokes an *influence* in the environment. An influence is an attempt to modify the course of events in the world. The $Communication$ module takes care for the communicative interactions. We developed a communication module that processes incoming messages and produces outgoing messages according to well-defined communication protocols [19]. Communication enables agents to exchange information, and set up collaborations reflected in mutual situated commitments.

Ongoing activities. Next to agents, we introduced the concept of an ongoing activity to model other processes in the system. An ongoing activity produces influences according to the state of the environment. Examples of ongoing activities are a moving object or an evaporating pheromone. Ongoing activities enable indirect coordination between agents, and as such form a basis for collective behavior. [16] discusses ongoing activities in detail.

Environment. As for agents, the architecture of the environment models different concerns (communication, perception generation and action handling) as separate modules. The $MessageDelivering$ module of the environment handles message transport. The $PerceptGenerator$ module generates representations of the local state of the environment for the agents [20]. The $Collector$ module collects the influences produced by agents and ongoing activities and passes them to the $Reactor$ module. This latter calculates, according to a set of domain specific laws, the reaction, i.e. state changes in the environment and consumptions for the agents. For a thorough discussion on influences and reactions we refer to [16].

3. Architectural design of an automatic guided vehicle transportation system

In this section we illustrate the architectural design of an AGV transportation system. This application is investigated in an ongoing research project in close cooperation with Egemin, a manufacturer of automated warehouse systems[4]. First we introduce the application and briefly discuss the existing centralized approach. Next we discuss new autonomic quality requirements for the application and illustrate how we have applied the reference architecture for situated MASs to design a new decentralized solution that aims to meet these quality requirements.

3.1 AGV transportation system

An AGV transportation system uses unmanned vehicles (AGVs) to transport *loads* through a warehouse. Typical applications are repackaging and distributing incoming goods to various branches, or distributing manufactured products to storage locations. An AGV uses a battery as its energy source. AGVs can move through a warehouse, following a physical path on the factory floor, guided by a laser navigation system, or magnets or cables that are fixed in the floor.

Functionalities. The main functionality the AGV transportation system should perform is handling *transports*, i.e. moving loads from one place to another. Transports are generated by *client systems*. Client systems are typically business management programs, but can also be particular machines, employees or service operators. A transport is composed out of multiple *jobs*: a job is a simple task that can be assigned to an AGV. 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 have to be assigned to AGVs that can execute them; (2) routing: AGVs must route efficiently through the layout of the

[4] http://www.egemin.com/home.html

warehouse when executing their transports; the best route for the AGVs in general is dynamic, and depends on the current conditions in the system; (3) collision avoidance: obviously, AGVs may 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. And finally, (4) deadlock prevention: the system must ensure that at least one of the necessary conditions for deadlock can never hold.

When an AGV is idle it can park at a neighboring park location; however, when the AGV runs out of energy, it has to charge its battery at one of the charging stations.

3.2 Traditional approach

Traditionally, vehicles are controlled by one central server, using wireless communication. This server has global knowledge of the system. It plans routes for AGVs according to incoming transports and instructs AGVs to perform the jobs. The server continuously polls the AGVs about their status. The low-level control of the AGVs in terms of sensors and actuators (staying on track on a segment, turning, and determining the current position, etc.), is handled by the AGV control software called E'nsor[5]. To this end, the layout of the factory is divided into logical elements: *segments* and *nodes*. A logical segment typically corresponds to a physical part of a path of three to five meters. E'nsor can be instructed to drive the AGV over a given segment; to drive the AGV over a given segment and pick up –or drop– a load at the end of it; to drive the AGV over a given segment to a battery charging station and start charging; and finally to drive the AGV over a given segment and park at the end of it.

New quality requirements. The evolution of the market put forward new requirements for AGV transportation systems. Customers request for flexibility of the transportation systems, AGVs should adapt their behavior with changing circumstances. Customers have various requests with respect to flexibility and adaptability, we discuss briefly a number of desired properties. AGVs should be able to exploit opportunities, e.g., when an AGV is assigned a transport and moves toward the load, is should be possible for this AGV to switch tasks on its way if a more interesting transport pops up. AGVs should also be able to anticipate possible difficulties, e.g., when the battery 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 exceptional situations, e.g., when a segment is blocked, the AGVs should avoid that segment.

In summary, flexibility and adaptability are high-ranking quality requirements for today AGV transportation systems.

3.3 A decentralized approach with situated MAS

We now present a decentralized solution of the AGV transportation system that aims to meet the quality requirements of flexibility and adaptability. Vehicles then become autonomous agents which make decisions based on their current knowledge, and who coordinate with other agents to ensure the system as a whole processes transports in time.

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

In this section we follow a trace in the architectural design of the situated MAS. We focus mainly on the module view of the architecture. In each step we refine one particular module of the architecture and motivate the main architectural decisions.

STEP 1: System decomposition as a situated MAS. At the top-level, the AGV transportation system is modelled as a situated MAS.

Motivation. Since decision making in a situated MAS is decentralized (the agents decide for themselves, locally), situated agents are able to react flexibly to different situations and adapt their behavior to changing circumstances. These properties make situated MAS a valuable candidate to cope with the target quality requirements. On the other hand, decentralization of control introduces additional complexity. In the MAS approach there is no repository of global knowledge available to resolve conflicts, AGVs have to coordinate among themselves. To avoid a communication bottleneck, communication must be localized as much as possible. And last but not least, a decentralized setup is harder to debug. The challenge in this project was (and is) to guarantee existing functionality, while aiming to realize the listed advantages by using a MAS based approach.

Fig. 2 depicts a high-level model view of the architecture of the MAS solution. The situated MAS consists of an environment and

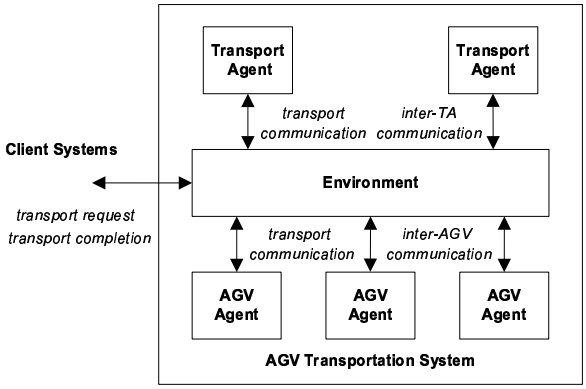

Figure 2. Module view of the architecture.

two kinds of agents, *transport agents* and *AGV agents*. A transport agent represents a transport in the system, it is responsible to determine the priority of the transport, to assign the transport to an AGV and to ensure that the transport is completed correctly. The priority of a transport depends on the kind of transport, the pending time since its creation, and the nature of other transports in the system. Therefore, transport agents interact with other related transport agents to determine the correct priority over time. AGV agents are responsible for executing the assigned transports. The environment provides communication infrastructure that enables agents to exchange information and to coordinate their behavior. The physical infrastructure, i.e., the AGVs equipped with sensors and actuators, the floor infrastructure to guide AGVs through the factory, and the loads that AGVs have to transport are also part of the environment. In the next step, we look at the architecture of the environment, in the following steps we elaborate on the architecture of the AGV agents.

STEP 2: Environment architecture. We first zoom in on the architecture of the environment that is set up as a layered architecture.

Motivation. To cope with the complexity of the environment, we have selected layers as architectural pattern for the architecture. Layers separate functionality, and support reuse.

Fig. 3 depicts the model view of the architecture of the environment. AGV agents and transport agents are situated in a virtual

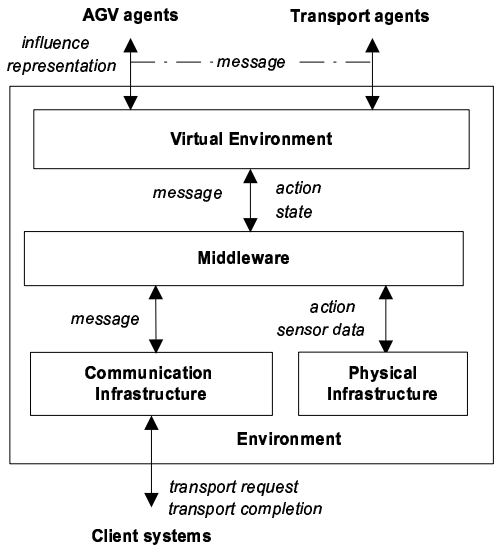

Figure 3. Architecture of the environment.

environment that is located at the top of the layered architecture. The virtual environment uses the middleware layer, that is composed of a message transfer system that enables agents to communicate with each other, the ObjectPlaces middleware [12] and E'nsor. The bottom layer of the environment consists of the infrastructure for communication and the physical infrastructure of the AGV transportation system. We now motivate and clarify the use of a virtual environment. The goal of the ObjectPlaces middleware is discussed in the next paragraph. For a detailed study of the virtual environment, we refer to [18].

Since the physical environment of a factory is very constrained, it restricts how agents can use their environment. We introduced a virtual environment for the agents to live in. This virtual environment offers a medium that agents can use to exchange information and coordinate their behavior. The use of the virtual environment is illustrated in Fig. 4. For clarity, we have simplified the explanation, for details see [18]. First we look how AGV agents coordinate through the virtual environment to avoid collisions. 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 then describes the physical area an AGV occupies along a certain path, see Fig. 4. 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. Afterwards, the AGV removes the markings in the virtual environment. Another use of the virtual environment are road signs. At each node in the layout, a sign in the virtual environment repre-

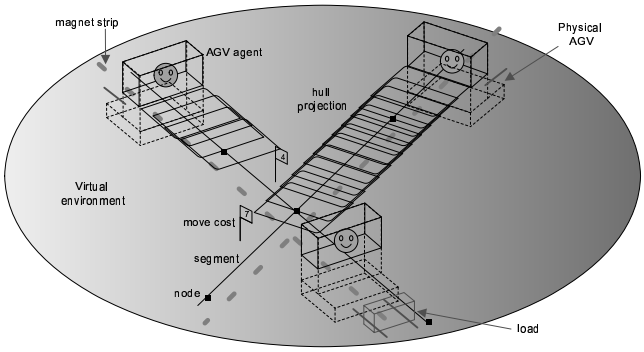

Figure 4. A virtual environment for AGV agents.

sents the cost to a given destination for each outgoing segment, see Fig. 4. 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 their environment, and uses them to determine which segment it will take next. Transport agents use the virtual environment to find AGV agents to assign the transports, and to follow the progress of the assigned transports. To assign the transport, the transport agent negotiates with AGV agents of idle AGVs near to the location of the load. Once the transport is assigned, the awarded AGV handles the transport.

Besides a medium for coordination, the virtual environment also serves as a suitable abstraction that shields the AGV agents from low-level issues, such as the physical control of the AGV. As part of the middleware, we fully reused the E'nsor software that deals with the low-level control of the AGVs. As such, the AGV agents control the movement and actions on a fairly high level.

Deployment view of the environment. We now explain how the virtual environment is deployed in the system, and what the role is of the ObjectPlaces middleware.

Fig. 5 depicts the deployment view of the AGV transportation system. A deployment view shows how the software is assigned to hardware processing and communication elements. Transport agents are located at *transport bases*. AGV agents are located in AGVs that are situated on the factory floor. Each AGV and each transport base is equipped with a processor. Communication infrastructure provides a wired network that connects client systems and transport bases, and a wireless network that enables mobile AGVs to communicate with each other and with transport agents. To avoid overload of this network, agents typically communicate only with other agents in their neighborhood.

Since the only physical infrastructure available to the AGVs is a wireless network to communicate, the virtual environment is necessarily distributed. In effect, each agent in the system maintains a *local virtual environment*, which is a local manifestation of the virtual environment. Synchronization of the state of the local virtual environment with neighboring neighboring agents is supported by the ObjectPlaces middleware. The local virtual environment uses the middleware by sharing objects in a tuplespace-like container, called an *objectplace*. Every AGV has one objectplace locally available. Objects in objectplaces on remote AGVs can be gathered using a *view*. A view specifies (1) which AGVs' object-

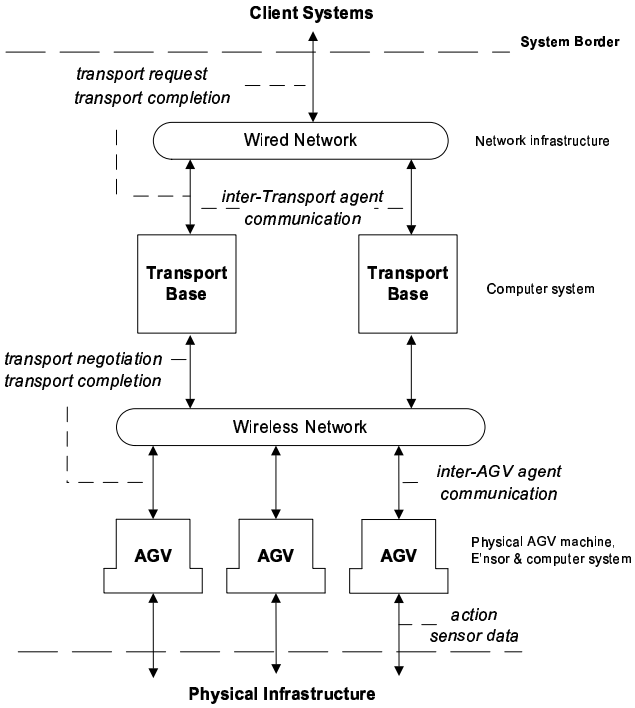

Figure 5. Deployment view of the system architecture.

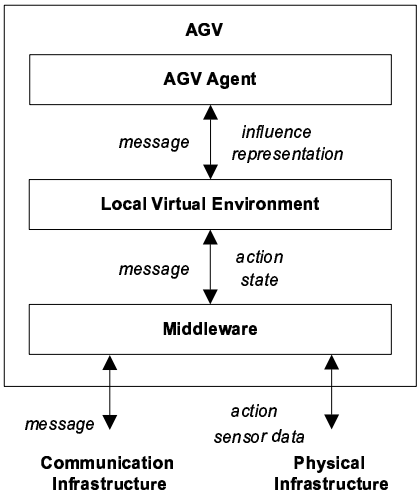

Figure 6. Software architecture deployed on AGVs.

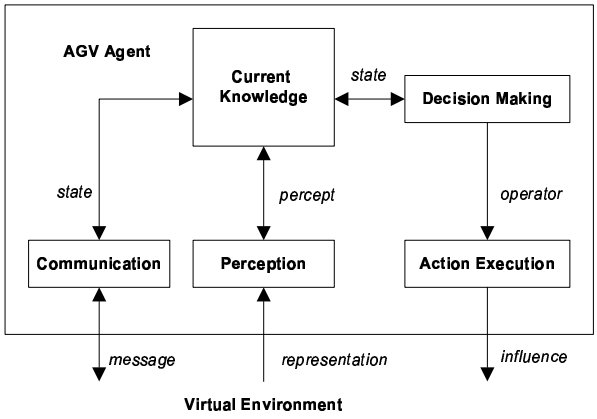

Figure 7. The AGV agent architecture.

places need to be included in the view (e.g. the objectplaces of all AGVs within a specific range), and (2) what objects need to be included in the view (e.g. hull objects). Fig. 6 depicts the architecture of the software that is deployed on each AGV. The AGV agent is shown in the top layer of the model. The two other layers correspond to the two top layers of the architecture of the environment, see Fig. 3.

STEP 3: AGV agent architecture, a data repository pattern. Now, we zoom in on the AGV agent. The AGV agent architecture corresponds to the agent architecture of the reference architecture and is basically modelled as a data repository pattern.

Motivation. The data repository pattern supports separation of concerns and loose coupling.

Fig. 7 depicts the module view of the AGV agent. Different concerns of the agent behavior, i.e., perception, communication, and decision making, are modelled as a separate modules of the architecture. The current knowledge module serves as data repository for the different modules.

STEP 4: Decision making, a combination of blackboard and sequential processing. To conclude, we zoom in on the decision making module. The decision making module is set up as a hybrid architecture that combines a blackboard pattern with sequential processing.

Motivation. This architecture combines complex interpretation of data with decision making at subsequent levels of abstraction.

The architecture of the decision making module is depicted in Fig. 8. The current knowledge module serves as blackboard data structure, while the action controller coordinates the selection of an appropriate operator. After job selection, the action selection module selects an action at a fairly high level (move, pick, park etc.). The action selection module is modelled as a free-flow architecture. Free-flow architectures allow to model flexible action selection. In [14], we have described a design process for free-flow architectures in detail. The action generation module transforms this action into a concrete preliminary operator (e.g., move(segment x)). The collision avoidance module is responsible to lock the trajectory associated with the selected operator. As soon as the trajectory is locked, the collision avoidance module passes the confirmed operator to the action execution module. If during the subsequent phases the selected operator can not be executed (e.g., an obstacle is detected on the selected trajectory), feedback is sent to the action controller that will inform the appropriate module to revise its decision. This feedback loop enables flexible decision making.

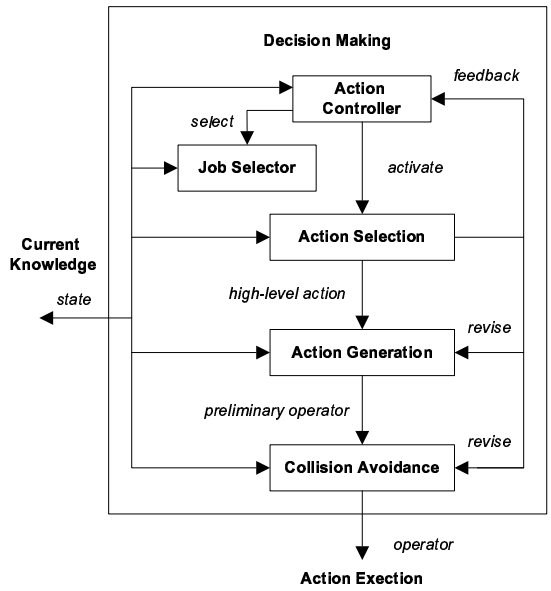

Figure 8. The Decision module.

4. Conclusion

In this paper, we argued that autonomic systems are essentially characterized by quality requirements that specify that the system should be able to adapt itself under varying circumstances and situations. We identified dynamism and change, and complexity, as important properties of problem domains when building autonomic systems. Target quality properties for an architecture to cope with dynamism and change are flexibility, adaptability and openness. Target quality properties to cope with complexity are manageability and unambiguity.

We have put forward situated MASs as an approach to build decentralized applications with autonomic quality requirements. In this paper, we discussed the architectural design of a self-managing AGV transportation system. In this application, AGVs should be able to exploit opportunities, anticipate possible difficulties, and cope with exceptional situations. Primary quality requirements to enable such self-managing behavior are flexibility and adaptability. We illustrated how the architectural decisions are guided by a reference architecture for situated MASs on the one hand, and architectural patterns that aim to satisfy the system's requirements on the other hand.

So far, we have validated the solution in an industrial test setup with two physical AGVs that execute predefined batches of jobs. Important lessons we learned from this initial project phase are: (1) the architecture is a great instrument for communication between different stakeholders; (2) in general, the reference architecture for situated MASs turned out to be an excellent guide for the architectural design; (3) the complexity of the application forced us to further decompose several modules of the reference architecture; (4) the concurrency view and the deployment view are as essential as the modular view to build quality software for a complex problem such as the AGV transportation system.

As future work, we intend to formalize the initial framework for interpreting autonomic quality requirements presented in this paper. This formalization will: (1) allow us to rigorously describe the decomposition of the non-functional requirements, and (2) serve as a reference for practical evaluation. The next challenges in the project are order assignment and deadlock avoidance, and subsequently the validation of the integral solution in an advanced setup.

5. References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.

[2] E. Bonabeau, F. Hnaux, S. Gurin, D. Snyers, P. Kuntz, and G. Theraulaz. Routing in telecommunications networks with ant-like agents. *IATA*, 1998.

[3] R. A. Brooks. Intelligence without representation. *Artificial Intelligence Journal*, 47, 1991.

[4] N. Chase. An autonomic computing roadmap, 2004. In *www-128.ibm.com/developerworks/library/ac-roadmap/*.

[5] J. Ferber. *An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, 1999.

[6] A. Ganek and T. Corbi. The dawning of the autonomic computing era. *Autonomic Computing*, 42(1), 2003.

[7] M. R. Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmanns, 1997.

[8] J. Kephart and D. Chess. The vision of autonomic computing. *Computer Magazine, IEEE*, 36(1), 2003.

[9] P. Maes. Modeling adaptive autonomous agents. *Artificial Life Journal*, 1(1-2), 1994.

[10] V. Parunak. The AARIA Agent Architecture: From manufacturing requirements to agent-based system design. *Integrated Computer-Aided Engineering*, 8(1), 2001.

[11] J. Sauter and H. Parunak. Ants in the supply chain. *Agent based Decision Support for Managing Supply Chains*, 1999.

[12] K. Schelfthout and T. Holvoet. Objectplaces: An environment for situated MASs. *3th Joint Conference on Autonomous Agents and Multi-Agent Systems*, 2004.

[13] M. Shaw and D. Garlan. Software architecture: perspectives on an emerging discipline. *Prentice-Hall*, 1996.

[14] E. Steegmans, D. Weyns, T. Holvoet, and Y. Berbers. A design process for adaptive behavior of situated agents. *Agent-Oriented Software Engineering, LNCS*, 3382, 2005.

[15] T. Tyrrell. Computational mechanisms for action selection. *University of Edinburgh*, 1993.

[16] D. Weyns and T. Holvoet. Formal model for situated MASs. *Fundamenta Informaticae*, 63(2), 2004.

[17] D. Weyns, H. Parunak, F. Michel, T. Holvoet, and J. Ferber. Environments for multiagent systems, state-of-the-art and research challenges. *LNCS*, 3374, 2005.

[18] D. Weyns, K. Schelfthout, and T. Holvoet. Exploiting a virtual environment in a real-world application. *Second Int. Workshop on Environments for Multiagent Systems*, 2005.

[19] D. Weyns, E. Steegmans, and T. Holvoet. Protocol based communication for situated MASs. *3th Joint Conference on Autonomous Agents and Multi-Agent Systems*, 2004.

[20] D. Weyns, E. Steegmans, and T. Holvoet. Towards active perception in situated multiagent systems. *Journal on Applied Artificial Intelligence*, 18(8-9), 2004.

[21] M. Wooldridge. *An Introduction to Multiagent Systems*. John Wiley and Sons, Ltd., England, 2002.