An Architectural Strategy for Self-Adapting Systems

Danny Weyns and Tom Holvoet
DistriNet Labs, Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{danny.weyns, tom.holvoet@cs.kuleuven.be}@cs.kuleuven.be

Abstract

Self-adaptation is the ability of a software system to adapt to dynamic and changing operating conditions autonomously. In this paper, we present an architectural strategy for self-adapting systems. An architectural strategy embodies architectural knowledge about a particular solution approach. The architectural strategy for self-adapting systems structures the software into a number of interacting autonomous entities (agents) that are situated in an environment. It integrates a set of architectural patterns that have proved to be valuable in the design of various self-adapting applications. The self-adapting properties of the approach are based on the agents' abilities to adapt their behavior to dynamic and changing circumstances. The architectural strategy provides an asset base architects can draw from when developing new self-adapting applications that share its common base.

1. Introduction

The advances in computing and communication technology pave the way to large-scale distributed software systems such as automatic traffic systems and large scale sensor networks. At the same time, this evolution introduces increasing levels of complexity to software engineers. Two important factors for the growing complexity are: the highly dynamic operating conditions under which software systems have to operate, and the inherent distribution of resources which makes central control practically infeasible. In our research, we investigate and apply situated multiagent systems for engineering such systems.

A situated multiagent system (situated MAS) structures the software in a number of interacting autonomous entities (agents) that are situated in an environment. Situated agents employ the environment to share information and coordinate their behavior. Control in situated MAS is decentralized, the system functionality results from the cooperation of the agents via the mediating environment. Decentralized control is essential to cope with the inherent distribution of resources. Situated MAS provide a way to model self-managing systems. 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 subject 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 [11].

Over the last five years, we have developed various situated MAS applications, ranging from a prototypical peer-to-peer file sharing system up to an industrial automated transportation [24, 4]. In the course of building these applications, we have developed an integrated set of architectural patterns for situated MAS. We call this set of patterns an *architectural strategy*. The architectural strategy embodies architectural knowledge about a particular solution approach. The self-adapting properties of the approach are based on the agents' abilities to adapt their behavior to dynamic and changing circumstances. The architectural strategy supports the development of new software architectures for self-adapting applications that share its common base.

Overview. The remainder of this paper is structured as follows. In section 2, we explain the notion of architectural strategy and show how an architectural strategy is related to other types of architectural approaches. Section 3 gives a high-level overview of the architectural strategy for self-adaptive systems. Section 4 compares our approach with the architectural blueprint for autonomic computing proposed by IBM. Finally, section 5 draws conclusions.

2. Architectural Strategy

Self-management is an increasingly important requirement for software intensive systems. Essentially, self-management is a quality property of a software system; it

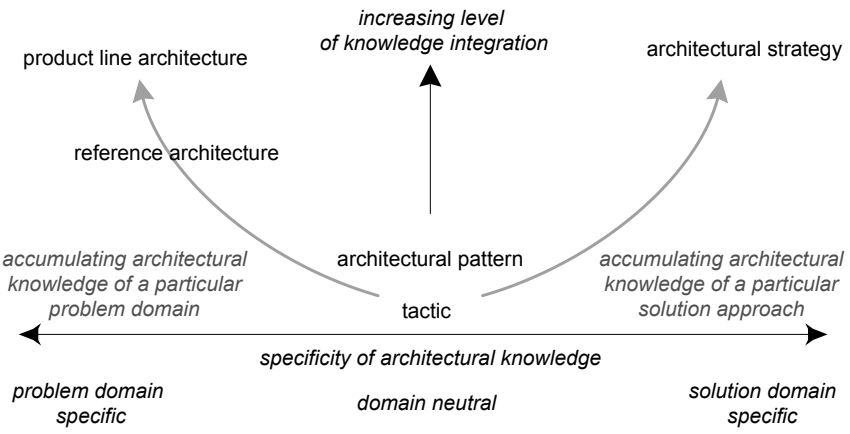

Figure 1. Types of architectural approaches.

allows a software system to adapt to dynamic and changing circumstances autonomously, keeping the complexity hidden to the user. In this section, we give a brief overview of architectural approaches for achieving qualities in general and then we put architectural strategy in this picture.

2.1. Architectural Approaches

The achievement of a system's quality requirements is based on design decisions. A *tactic* is a widely used architectural approach that has proven to be useful to achieve a particular quality [2, 15]. For example, "rollback" is a tactic to recover from a failure aiming to increase a system availability. Actually, to realize one or more tactics an architect typically chooses an appropriate *architectural pattern / style* [17]. An architectural pattern is "a description of architectural elements and relation types together with a set of constraints on how they may be used" [2]. An architectural pattern is a recurring architectural approach that exhibits particular quality attributes. Examples of common architectural patterns are layers, pipe-and-filter, and blackboard. Reference architectures [13, 2] go one step further in reuse of best practices in architectural design. The Rational Unified Process [12] 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 with specific functionality and quality attribute requirements. The concept of a reference architecture is closely related to a product line architecture. A product line architecture provides a set of software-intensive systems sharing a common, managed set of features that satisfy specific needs of a particular market segment and that are developed from a common set of core assets in a prescribed way [8]. A product line architecture stresses strategic reuse of architectural assets in producing a family of products.

2.2. Architectural Strategy

An architectural strategy integrates a set of architectural patterns that have proved to be useful in architectural design, similar to a reference architecture. Yet, both architectural approaches differ in the type of architectural knowledge they represent. Fig. 1 shows an overview of different architectural approaches. Tactics and architectural patterns provide reusable architectural building blocks to achieve particular quality attributes. Reference architectures and product line architectures integrate architectural building blocks, embodying architectural knowledge about a particular *problem* domain. An architectural strategy complements these approaches by embodying architectural knowledge about a particular *solution* approach. We now compare the different approaches in more detail.

Architectural Patterns. Fig. 2 shows the layers pattern. The architectural elements of this pattern are layers that are related by the "is allowed to use" relationship. Layers are neutral with respect to problem or solution domains.

Product Line Architecture. Fig. 3 shows an excerpt from a product line architecture of a satellite control system. The components in this view are organized according to a number of patterns. Maneuver, Orbit, Attitude, Vehicle, etc. are components that refer to concepts that are specific to the problem domain. The components in the shaded area covers core components of the product line architecture. These components provide common features of the product line and support variation to develop specific products.

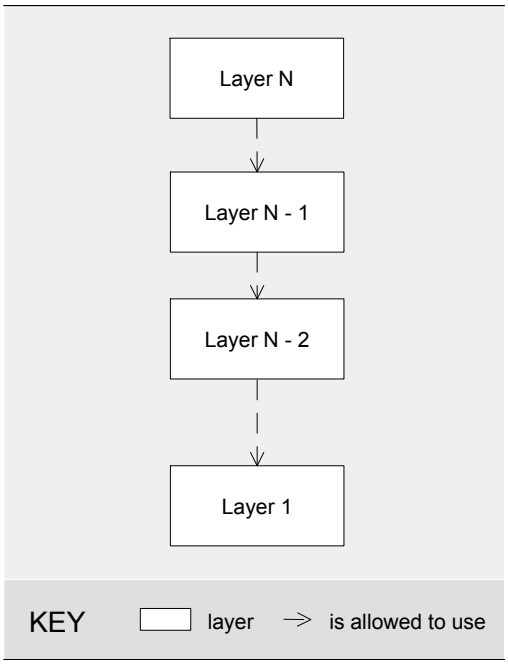

Figure 2. Layers pattern [17, 6]

Architectural Strategy. Fig. 5 shows an excerpt of the architectural strategy for situated MAS. The elements in this architectural view refer to concepts in the solution domain of situated MAS. An Agent is an autonomous entity that uses Perception to sense the environment, Decision Making to act in the environment, and Communication to interact with other agents.

An architectural strategy provides architectural knowledge to develop software systems with particular characteristics and system requirements. On the one hand, the architectural strategy defines constraints on the architectural elements and their relations, on the other hand, it defines the required variability to develop concrete applications. For example, Agent in the architectural strategy is abstractly defined. Yet, for a concrete application different agent types have to be defined that have different responsibilities which will be reflected in different behavior and internal architectural structures. For example, for an automated transportation system that uses automatic guided vehicles (AGVs), each AGV may be controlled by an AGV agent, while each transport may be represented by a Transport agent that is responsible to assign the transport to an AGV.

3. Architectural Strategy for Self-Adapting Systems

We now give an overview of the architectural strategy for situated multiagent systems. The target domain of the architectural strategy are software systems: (1) that are subject to highly dynamic and changing operating conditions, such as dynamically changing workloads and variations in availability of resources and services; (2) in which 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. Example domains are mobile and ad-hoc networks, sensor networks, and automated transportation and traffic control systems. Important quality properties of the architectural strategy are flexibility and openness. With flexibility we refer to the system's ability to deal with dynamic operating conditions, while openness refers to the system's ability to deal with changing situations, such as agents that enter and leave the system.

We have documented the architectural strategy for self-adapting systems by means of various architectural views [7]. In this paper, we limit the overview to the main architectural views, i.e the top-level module decomposition of a situated MAS and two component and connector views. [20] gives an integral description of the architectural strategy, including a description of the variability points and a formal specification of the various architectural elements.

3.1. Top-Level Module Decomposition

Fig. 5 shows the top-level module decomposition of the architectural strategy that shows the main software units in the system. A situated MAS is decomposed in two basic modules: `Agent` and `Application Environment`.

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

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

The `Application Environment` enables agents to share information and to coordinate their behavior [23, 19]. 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.

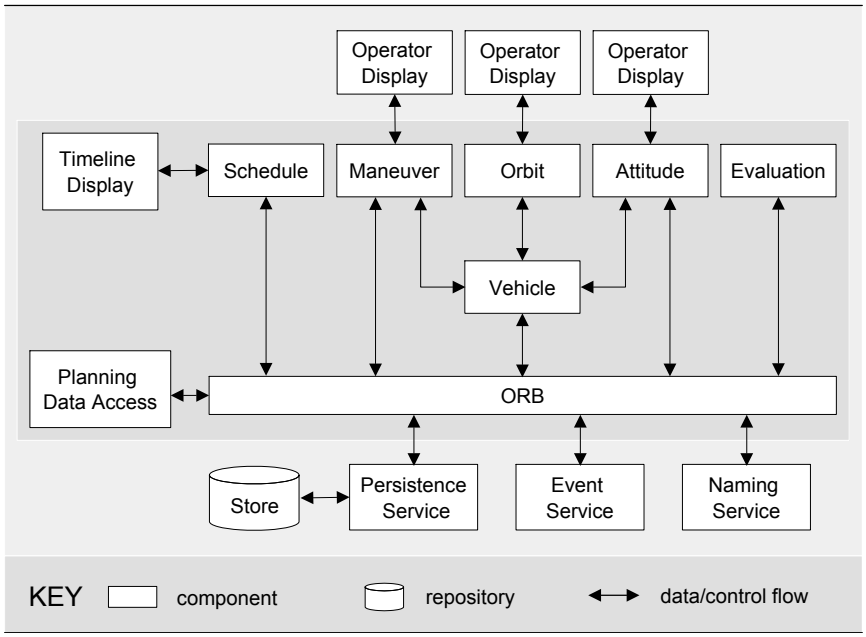

Figure 3. Excerpt from a product line architecture for satellite control [8]

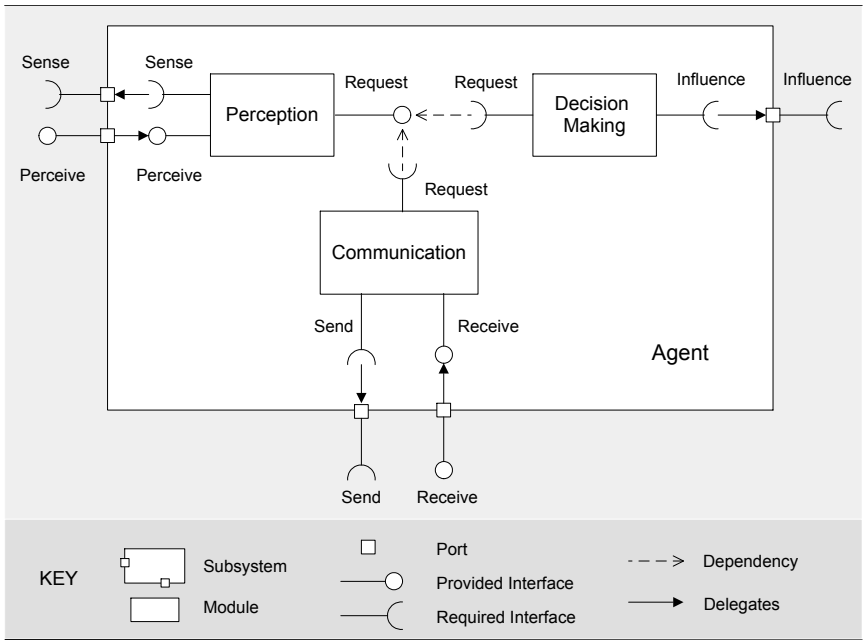

Figure 4. Excerpt of the architectural strategy for situated multiagent systems [20]

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

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

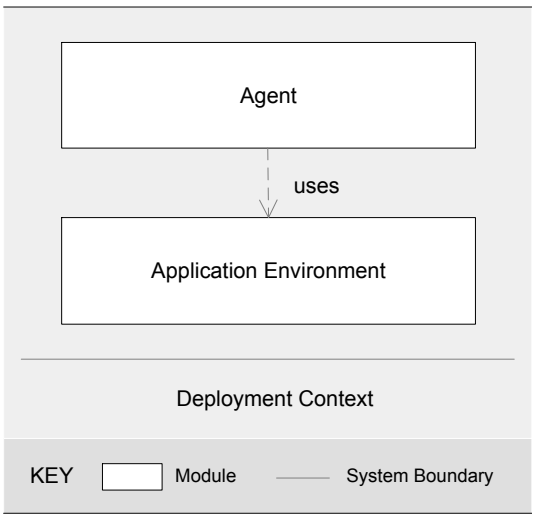

Figure 5. Top-level module decomposition

ture [5]. Such infrastructure enables agents to dynamically form paths to locations of interest mimicking the behavior of social insects.

For a distributed application, the deployment context consists of multiple processors deployed on different nodes that are connected through a network. Each node provides an application environment to the agents located at that node. Depending on the specific application requirements, different application environment types may be provided. For some applications, the same type of application environment subsystem is instantiated on each node. For other applications, specific types are instantiated on different nodes, e.g., when different types of agents are deployed on different nodes.

Rationale. The main principles that underly the decomposition of a situated MAS are:

- *Decentralized control.* In a situated MAS, 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 architectural strategy.

- *Self-management.* In a situated MAS 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 are important requirements of the target applications of the architectural strategy.

However, the decentralized architecture of a situated MAS 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 MAS. Special attention should be payed to communication which could impose a major bottleneck.

3.2 Collaborating Components View

The collaborating components view shows the MAS or parts of it as a set of interacting runtime components that use a set of shared data repositories to realize the required system functionalities [20]. The elements of the collaborating components view are: (1) runtime components that achieve a part of the system functionality, (2) data repositories that enable multiple runtime components to share data, (3) component–repository connectors that 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, (4) component–component connectors enable runtime components to request each other to perform a particular functionality.

The collaborating components view is an excellent vehicle to learn the runtime behavior of a situated MAS. The view shows the data flows between runtime components and the interaction with data stores, and it specifies the functionalities of the various components in terms of incoming and outgoing data flows. We discuss two view packets of the collaborating components view. We start with the view packet that describes the collaborating components of agent. Next, we discuss the view packet that describes the collaborating components of the application environment.

3.2.1 Collaborating Components of Agent

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

Elements and their Properties The `Agent` component (i.e. a runtime instance of the Agent module shown in Fig. 5) consists of three subcomponents: `Perception`, `Decision Making`, and `Communication`. These components share the `Current Knowledge` repository. We first give a brief explanation of the responsibilities of the

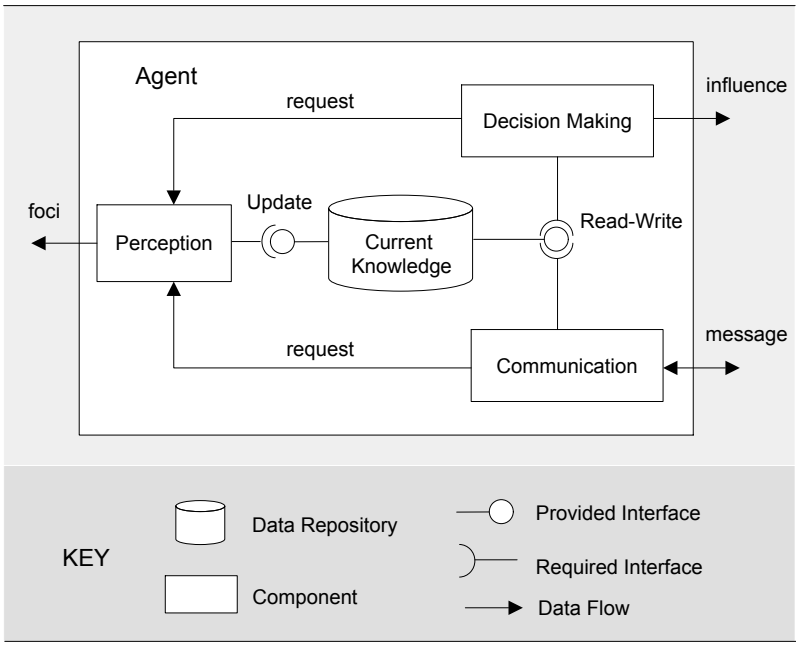

Figure 6. Collaborating Components of Agent

components and then we explain the architecture rationale of the view packet.

`Perception` is responsible for collecting runtime information from the environment (application environment and deployment context). The perception component supports selective perception [26]. Selective perception enables an agent to direct its perception according to its current tasks. To direct its perception an agent selects a set of foci and filters. Foci allow the agent to sense the environment only for specific types of information. Sensing results in a representation of the sensed environment. A representation is a data structure that represents elements or resources in the environment. The perception module maps this representation to a percept, i.e. a description of the sensed environment in a form of data elements that can be used to update the agent's current knowledge. The selected set of filters further reduces the percept according to the criteria specified by the filters.

`Decision Making` is responsible for action selection. The action model of the architectural strategy is based on the influence–reaction model introduced in [9]. 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 [21]. Situated agents use a behavior-based action selection mechanism [22]. Behavior-based action selection is efficient allowing agents to adapt their behavior quickly with changing circumstances. To enable situated agents to set up collaborations, we have extended behavior-based action selection mechanisms with roles and situated commitments [18, 25]. A role represents a coherent part of an agent's functionality in the context of an organization. A situated commitment is an engagement of an agent to give preference to the actions of a particular role in the commitment. Agents typically commit relative to one another in a collaboration, but an agent can also commit to itself, e.g. when a vital task must be completed. Roles and commitments have a well-known *name* that is part of the domain ontology and that is shared among the agents in the system. Sharing these names enable agents to set up collaborations via message exchange. We explain the coordination among decision making and communication below.

`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 [25]. A communication protocol specifies a set of possible sequences of messages. We use the notion of a *conversation* to refer to an ongoing communicative interaction. A conversation is initiated by the initial message of a communication protocol. At each stage in the conversation there is a limited set of possible messages that can be exchanged. Terminal states determine when the conversation comes to an end.

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

`Current Knowledge` repository contains data that is shared among the data accessors. Data stored in the current knowledge repository refers to state perceived in the environment, to state related to the agent's roles and situated commitments, and possibly other internal state that is shared among the data accessors. The current knowledge repository exposes two interfaces, see Fig. 6. 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.

Rationale. The overall behavior of the agent is the result of the coordination of two components: decision making and communication. To complete the agent's tasks, decision making and communication coordinate via the current knowledge repository. For example, agents can send each other messages with requests for information that enable them to act more efficient. 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 toward actions in the agent's role in the collaboration. This continues until the commitment is deactivated and the collaboration ends.

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

3.2.2 Collaborating Components Application Environment

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

Elements and their Properties The `Application Environment` component consists of seven subcomponents and the shared `State` repository. We discuss the responsibilities of each of the elements in turn. Then, we explain the architecture rationale of the view packet.

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 [26]. For example, for reasons of efficiency a designer can introduce default limits for perception in order to restrain the amount of information that has to be processed, or to limit the occupied bandwidth.

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

`Interaction` is responsible to deal with agents' influences in the environment. Agents' influences can be divided in two classes: influences that attempt to modify state of the application environment and influences that attempt to modify the state of resources of the deployment context. An example of the former is an agent that drops a digital pheromone in the environment. An example of the latter is an agent that writes data in an external data base. Agents' influences are subject to action laws [21]. Action laws put restrictions on the influences invoked by the agents, representing domain specific constraints on agents' actions. For example, when several agents aim to access an external resource simultaneously, an interaction law may impose a policy on the access of that resource. For influences that relate to the application environment, the interaction component calculates the reaction of the influences resulting in an up-

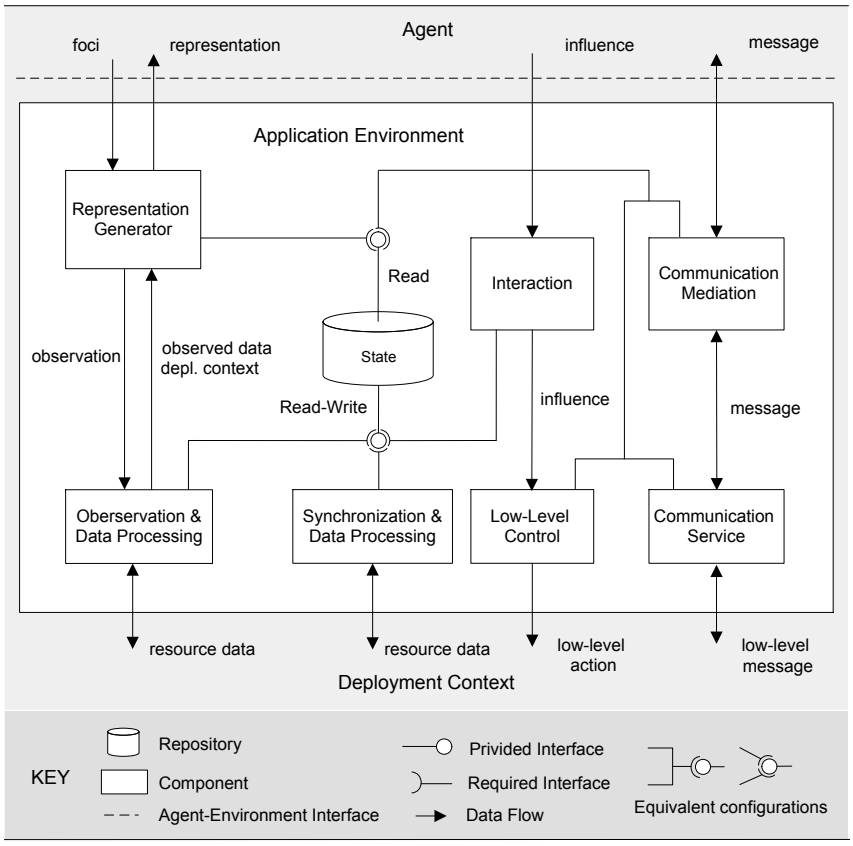

Figure 7. Collaborating Components of Application Environment

date of the state of the application environment. Influences related to the deployment context are passed to the Low-Level Control component.

`Low-Level Control` converts the influences invoked by the agents into low-level action primitives in the deployment context. This decouples the interaction component from the details of the deployment context.

`Communication Mediation` mediates the communicative interactions among agents. It is responsible for collecting messages; it provides the necessary infrastructure to buffer messages, and it delivers messages to the appropriate agents. Communication mediation regulates the exchange of messages between agents according a set of applicable communication laws [25]. Communication laws impose constraints on the message stream or enforce domain–specific rules to the exchange of messages. Examples are a law that drops messages directed to agents outside the communication–range of the sender and a law that gives preferential treatment to high-priority messages. To actually transmit the messages, communication mediation makes use of the Communication Service component.

`Communication Service` provides that actual infrastructure to transmit messages. Communication service transfers message descriptions used by agents to communication primitives of the deployment context. For example, FIPA ACL message [10] enable a designer to express the communicative interactions between agents independently of the applied communication technology. However, to actually transmit such messages, they have to be translated into low-level primitives of a communication infrastructure provided by the deployment context. Depending on the specific application requirements, the communication service may provide specific communication services to enable the exchange of messages in a distributed setting, such as white and yellow page services. An example infrastructure for distributed communication is Jade [3]. Specific middleware may provide support for communicative interaction in mobile and ad-hoc network environments, an example is discussed in [16].

`Synchronization & Data Processing` synchronizes state of the application environment with state of resources in the deployment context as well as state of the application environment on different nodes. State updates may relate to dynamics in the deployment context and dynamics of state in the application environment that happens

independently of agents or the deployment context. An example of the former is the topology of a dynamic network which changes are reflected in a network abstraction maintained in the state of the application environment. An example of the latter is the evaporation of digital pheromones. Middleware may provide support to collect data in a distributed setting. An example of middleware support for data collection in mobile and ad-hoc network environments (views) is discussed in [14]. Synchronization & data processing converts the resource data observed from the deployment context into a format that can be used to update the state of the application environment. Such conversion typically includes a processing of collected resource data.

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 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. Besides influences invoked by agents, we consider activity from the deployment context that affects the state of the application environment (synchronization & data processing) 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 mediation, and interaction. The top row deals with the mediation of agents activities in the system. The bottom row deals with the interaction of the application environment with the deployment context and consists of observation & data processing, low-level control and synchronization & data processing, and communication service.

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 mediation and communication service) can be omitted.

Each module of the application environment is located in a particular column and row and is assigned a particular functionality. Minimizing the overlap of functionality among modules, helps the architect to focus on one particular aspect of the functionality of the application environment. It supports reuse, and it further helps to accommodate change and to update one module without affecting the others.

4. Comparison with IBM Blueprint Architecture for Autonomic Computing

In this section, we reflect on the architectural blueprint for autonomic computing proposed by IBM and we point to opportunities provided by the architectural strategy for situated MAS to the blueprint architecture.

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 [11]. IBM has developed an architectural blueprint for autonomic computing. This architectural blueprint specifies the fundamental concepts and the architectural building blocks used to construct autonomic systems [1]. 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.

Although presented as architecture, to our opinion, the blueprint describes a reference model. The discussion mainly focuses 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 architectural strategy for self-adapting 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.

5. Conclusions

In this paper, we gave a high-level overview of an architectural strategy for self-adapting systems. This architectural strategy generalizes common functions and structures from various experimental applications we have studied and built. This generalized architecture provides a reusable design artifact, it embodies architectural knowledge that allows architects to design new software architectures for systems that share the common base more reliably and cost effectively.

The current description of the architectural strategy allows flexible modeling of architectural elements and relations; however, the documentation exhibits semantic incompleteness. This hampers reasoning on an architecture and verification of system properties. Our long-term goal is to develop a formally founded architectural description language (ADL) for decentralized, self-adapting systems. Such an ADL enables analysis and formal verification of desired system properties which is crucial for the practical applicability of such systems.

References

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

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley Publishing Comp., 2003.

[3] 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.

[4] 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, Lecture Notes in Computer Science, Vol. 4214*, Vasteras, Sweden, 2006. Springer.

[5] S. Brueckner. *Return from the Ant, Synthetic Ecosystems for Manufacturing Control*. Ph.D Dissertation, Humboldt University, Berlin, Germany, 2000.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Patten-Oriented Software Architecture*. John Wley and Sons, November 2002.

[7] 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.

[8] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Publishing Comp., August 2001.

[9] 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.

[10] FIPA. Foundation for Intelligent Physical Agents, FIPA Abstract Architecture Specification. *http://www.fipa.org/repository/bysubject.html*, (8/2006).

[11] J. Kephart and D. Chess. The Vision of Autonomic Computing. *IEEE Computer Magazine*, 36(1):41–52, 2003.

[12] P. Kruchten. *The Rational Unified Process*. Addison Wesley Publishing Comp., 2003.

[13] P. Reed. Reference Architecture: The Best of Best Practices. *The Rational Edge*, 2002. www-128.ibm.com/developerworks/rational/library/2774.html.

[14] G. Roman, C. Julien, and A. Murphy. A Declarative Approach to Agent Centered Context-Aware Computing in Ad Hoc Wireless Environments, Software Engineering for Large-Scale Multi-Agent Systems, Lecture Notes in Computer Science, Vol. 2603, 2003.

[15] N. Rozanski and E. Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison Wesley Publishing Comp., 2005.

[16] K. Schelfthout, D. Weyns, and T. Holvoet. Middleware that Enables Protocol-Based Coordination Applied in AGV Control. *IEEE Distributed Systems Online*, 7(8), 2006.

[17] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[18] E. Steegmans, D. Weyns, T. Holvoet, and Y. Berbers. A Design Process for Adaptive Behavior of Situated Agents. In *Agent-Oriented Software Engineering V*, Lecture Notes in Computer Science, Vol. 3382. Springer, 2004.

[19] M. Viroli, T. Holvoet, A. Ricci, K. Schelfthout, and F. Zambonelli. Infrastructures for the Environment of Multiagent Systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):49–60, 2007.

[20] D. Weyns. An Architecture-Centric Approach for Software Engineering with Situated Multiagent Systems. *Ph.D, Katholieke Universiteit Leuven*, 2006.

[21] D. Weyns and T. Holvoet. Formal Model for Situated Multi-Agent Systems. *Fundam. Inform.*, 63(1-2):125–158, 2004.

[22] 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*, Lecture Notes in Computer Science, Vol. 3963. Springer-Verlag, 2006.

[23] D. Weyns, A. Omicini, and J. Odell. Environment as a First-Class Abstraction in Multiagent Systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–29, 2007.

[24] 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.

[25] 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.

[26] D. Weyns, E. Steegmans, and T. Holvoet. Towards Active Perception in Situated Multi-Agent Systems. *Applied Artificial Intelligence*, 18(9-10):867–883, 2004.