Multiagent Systems as Software Architecture

Another Perspective on Software Engineering with Multiagent Systems

Danny Weyns
Katholieke Universiteit Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
danny@cs.kuleuven.be

Tom Holvoet
Katholieke Universiteit Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
tom@cs.kuleuven.be

Kurt Schelfthout
Katholieke Universiteit Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
kurts@cs.kuleuven.be

ABSTRACT

The trend in agent-oriented software engineering is to consider multiagent systems (MASs) as a *radically new* way of engineering software. This position isolates agent-oriented software engineering from mainstream software engineering and could be one important reason why MASs are not widely adopted in industry yet.

In this paper, we present another perspective on software engineering with MASs. We put forward MASs as *software architecture*. We give an overview of a reference architecture for situated MAS. This reference architecture extracts and generalizes common functions and structures from various applications we have studied and built. The reference architecture provides a blueprint for architectural design of MAS applications that share the come base of the systems it is derived from. Considering MASs essentially as software architecture paves the way to integration with mainstream software engineering.

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Multiagent systems; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design

1. INTRODUCTION

Researchers and practitioners in agent-oriented software engineering generally consider MASs as a *radically new* way of engineering software. Here is a list of recent quotes from literature:

- There is a fundamental mismatch between the concepts used by mainstream software engineering and the agent-oriented view. Existing software development techniques are unsuitable to realize the potential of agents as a software engineering paradigm. [15]
- Whether agent-oriented approaches catch on as a software engineering paradigm [will depend on] the degree to which agents represent a radical departure from current software engineering thinking. [5]
- 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. [17]
- 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. [16]

This vision has led to the development of numerous MAS methodologies. Some of the proposed methodologies adopt techniques and practices from mainstream software engineering such as object-oriented techniques, e.g. Prometheus [6]; the Rational Unified Process, e.g. Mase [14]; or the Unified Modeling Language, e.g. Adelfe [3]. However, nearly all of these methodologies start from the idea that MAS provides a new way to engineer software systems. The position of being a radically new paradigm for software development isolates agent-oriented software engineering from mainstream software engineering. This could well be one important reason why MASs are not widely adopted in industry yet.

This paper presents another perspective on software engineering with MASs. We put forward MASs as *software architecture*. Considering MASs as software architecture gives MAS a clear position in the software development process, and paves the way to integration with mainstream software engineering.

Overview. Section 2 explains our perspective of MAS as software architecture. In section 3, we give a brief overview of architectural design that provides a context to apply MAS as software architecture in software engineering. Section 4 gives an overview of a reference architecture for situated MAS that architects can use for the design of concrete software architectures. Finally, we conclude in section 5.

2. MULTIAGENT SYSTEMS AS SOFTWARE ARCHITECTURE

Instead of considering MASs as a radically new approach for software development, we aim to give MASs a position in a general software engineering process. Our perspective on the essential purpose of MASs is as follows:

Multiagent systems provide an approach to solve a software problem by decomposing the system into a number of autonomous entities embedded in an environment in order to achieve the functional and quality requirements of the system.

This perspective states that MASs are an approach to *solve* a software problem. In particular, a MAS is a specific *decomposition of interacting elements* of the system intended to achieve the *requirements* of the system. This is exactly what *software architecture* is about. [1] 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 provide the functionality of the system, while the required quality attributes are primarily achieved through the structures of the software architecture. As such, MASs are in essence a family—yet a large family—of software architectures. The elements of a MAS software architecture are agents, resources, services, environment, etc. Relationships among the elements are very diverse, ranging from indirect interaction through virtual pheromone trails to complex negotiation protocols. In short, MASs are a very rich family of architectural approaches with specific characteristics, useful for a diversity of challenging application domains. Based on the problem analysis that yields the functional and quality attribute requirements of the system, the architect may or may not choose for a MAS-based solution. Quality attribute requirements such as robustness, flexibility or openness may be arguments to choose for a MAS software architecture. As such, we consider MASs as one valuable family of approaches to solve software problems in a large spectrum of possible ways to solve problems.

3. ARCHITECTURAL DESIGN

We now give a brief overview of architectural design that provides a context to apply MASs as software architecture. In the software development life cycle, architectural design iterates with requirements analysis on the one hand, and with the detailed design and development of the system on the other hand. Architectural design includes the design, the documentation, and the evaluation of the software architecture.

Design. Architectural design requires a systematic approach to develop a software architecture that meets the system requirements. In our research, we use techniques from the Attribute Driven Design (ADD [2]) method to design the architecture for a software system. The ADD method is a recursive decomposition method that is based on understanding how to achieve quality goals through proven architectural approaches. One common architectural approach are architectural patters [7]. An architectural pattern is a description of architectural elements and their relationships that has proven to be useful for achieving particular qualities. Examples of architectural patterns are pipe–and–filter or blackboard. We have developed a reference architecture for situated multiagent systems as a reusable architectural approach. This reference architecture integrates a set of architectural patterns that have proven their value in various MAS applications we have studied and built. The reference architecture provides an asset base the architect can draw from to select suitable architectural solutions.

Documentation. A software architecture is described by *views* [1]. A view describes the architecture of a software system from a particular perspective. The main views are: the module view that documents a system's principal units of implementation; the component-and-connector view that documents the system's units of execution; and the deployment view that documents the relationships between a system's software and its development and execution environment. Documenting a software architecture comes down to documenting the relevant views of the software architecture for the application at hand.

Evaluation. A software architecture is the foundation of a software system, it represents a system's earliest set of design decisions. Due to its large impact on the development of the system, it is important to verify the architecture as soon as possible. Modifications in early stages of the design are cheap and easy to carry out. In our research, we use the Architectural Tradeoff Analysis Method (ATAM [8]). The ATAM incites the stakeholders to articulate and prioritize specific quality goals; it forces the architect to provide a clear explanation and documentation of the software architecture; and especially it uncovers problems with the architecture that can be used to improve the quality of the software architecture in an early stage of the development cycle.

4. REFERENCE ARCHITECTURE FOR SITUATED MULTIAGENT SYSTEMS

The reference architecture we present in this section generalizes and extracts common functions and structures from various applications we have studied and built, including the Packet-World, a P2P file sharing system, a number of basic robot applications, and an simulator for Automatic Guided Vehicle systems. Besides these basic applications, the reference architecture considerably draws from experiences with an industrial logistic transportation system for warehouses [11]. The reference architecture provides a blueprint for architectural design of MAS applications that share the come base of the systems it is derived from.

Fig. 1 shows an overview of the reference architecture. The architecture integrates two primary abstractions: *agents* and the *environment*.

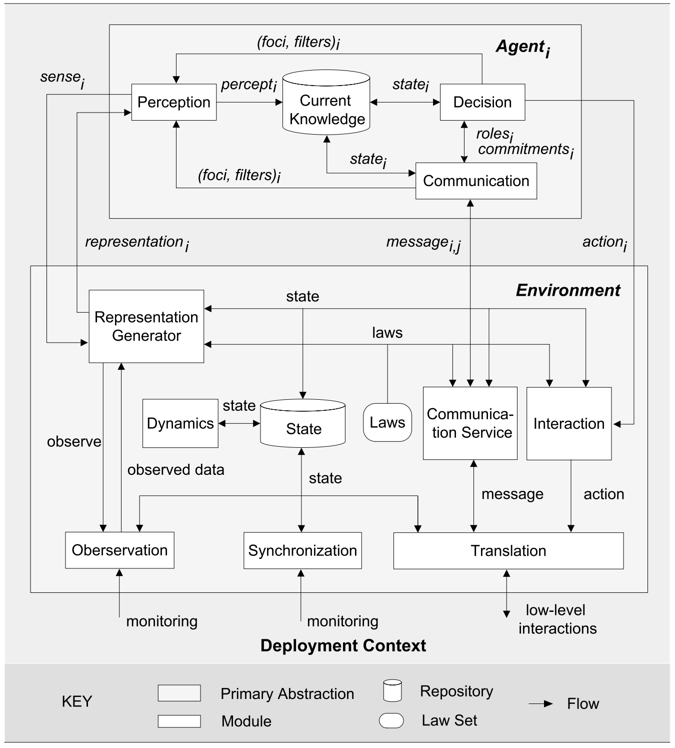

Figure 1: High-level overview of the reference architecture for situated multiagent systems

Agents. The agent architecture models perception, decision making and communication as separate modules of the agent. The *Perception* module maps a local representation of the state of the environment to a percept for the agent. We developed a model for *selective perception* that enables an agent to direct its perception at the most relevant aspects in the environment according to its current task. To sense its environment, the agent selects a set of *foci*. Foci allows the agent to sense the environment only for specific types of information. Sensing results in a representation of the agent's surrounding that can be interpret by the agent producing a percept. Finally, the percept is filtered by a set of selected *filters*, restricting the perceived data according to context relevant selection criteria. [13] zooms in on the perception module and discusses the different submodules in detail. The *Current Knowledge* module integrates percepts to update the knowledge of the agent.

The *Decision* module is responsible for action selection. We developed the decision module as a free-flow architecture, for details, we refer to [9]. 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 to one another via communication. Roles and situated commitments are building blocks for explicit collective behavior. The action selected by the decision module is passed to the environment for execution.

Communication enables agents to exchange information, and set up collaborations reflected in mutual situated commitments. We developed a communication module that processes incoming messages and produces outgoing messages according to well-defined communication protocols. The communication module (1) interprets messages and reacts to them according to the applicable protocol, and (2) initiates and continues conversations when the conditions imposed by the applicable protocol are satisfied. Messages are passed to the message delivering system of the environment. The communication module is discussed in detail in [12].

Environment. The environment offers functionality to the agents for perception, communication and action, and it provides an interface to the *deployment context*. With deployment context, we refer to the given hardware and software and external resources with which the MAS interacts such as sensors and actuators, a printer, a network, a database, or a webservice. We now explain the different modules of the environment, for a detailed discussion of the environment model see [10].

The *State* repository represents the actual state of the environment. The environment state typically includes an abstraction of the deployment context possibly extended with other state related to the MAS environment. An example of state related to the deployment context is an abstract representation of a network topology. An example of additional state is the representation of digital pheromones that overlays the network.

The *Synchronization* module monitors specific parts of the deployment context and keeps the corresponding representation in the state repository up to date. An example is a synchronization module that monitors the topology of a dynamic network and maintains the representation of the network structure in the state repository.

The *Dynamics* module maintains dynamics in the environment that happens independent of the agents or the deployment context. A typical example is the the evaporation of a digital pheromone.

Laws represent application-specific constraints on agents' perception, interactions, and communication. We discuss examples of different types of laws below.

The *Percept Generator* generates percepts for the agents. In general, agents can perceive state of the environment, or observe elements in the deployment context. In this latter case, the percept generator requests the *Observation* module to retrieve the required data form the deployment context. Perception is subject to laws that provide a means to constrain perception. 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. [13] discusses the perception generator in detail.

The *Interaction* module deals with agents' actions. Actions can be divided in two classes: actions that may result in a modification of state of the environment, and actions that may result in the modification of elements 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. Actions are subject to interaction laws. For example, when several agents aim to access an external resource, an interaction law may impose a policy on the access of that resource. Actions related to the deployment context are passed to the *Translation* module that converts the high-level actions of agents into low-level interactions in the deployment context.

The *Communication Service* module collects messages and delivers messages to the appropriate agents. Message delivering can be subject to communication laws. Communication laws can just regulate the message stream, or they can impose application-specific regulations on exchanged messages. An example of this latter are a set of laws that impose agents to follow the prescribed steps of a particular protocol. The translation module converts the high-level message descriptions into low-level communication primitives of the deployment context.

Industrial Application. We have used the reference architecture for situated MASs for the architectural design of an industrial system for logistics services in a tobacco warehouse. This real-world application uses automatic guided vehicles to transport loads in the warehouse [11]. For the documentation of the software architecture of this application and a detailed report of the ATAM evaluation we refer to [4].

5. CONCLUSION

In this paper, we presented another perspective on software engineering with MASs. We put forward MASs as software architecture. Looking upon MASs as software architecture does not dilute MAS as a software engineering paradigm, on the contrary, it gives MAS a clear and prominent role in the software development process, paving the way to integration with mainstream software engineering.

We gave an overview of a reference architecture for situated MASs. The reference architecture integrates a set of patterns that architects can use during architectural design of concrete MAS applications.

Intensive experience with applying the reference architecture in a complex industrial application has convinced us that the MAS community would benefit a great deal from allocating a correct place for MAS in mainstream software engineering.

6. REFERENCES

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

[2] L. Bass, M. Klein, and F. Bachmann. Quality Attribute Design Primitives and the Attribute Driven Design Method. *4th International Workshop on Product Family Engineering*, 2001.

[3] C. Bernon, M.-P. Gleizes, S. Peyruqueou, and G. Picard. Adelfe: A Methodology for Adaptive Multiagent Systems Engineering. *Volume 2577 of Lecture Notes in Computer Science*, 2002.

[4] N. Boucke, T. Holvoet, T. Lefever, R. Sempels, K. Schelfthout, D. Weyns, and J. Wielemans. Applying the ATAM to an industrial multi-agent system application. In *CW-431, Technical Report, K.U.Leuven, Belgium*.

[5] N. R. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4), 2001.

[6] L. Padgham and M. Winikoff. Prometheus: A Methodology for Developing Intelligent Agents. *Volume 2585 of Lecture Notes in Computer Science*, 2003.

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

[8] Software Engineering Institute. Carnegie Mellon University. *http://www.sei.cmu.edu/*.

[9] E. Steegmans, D. Weyns, T. Holvoet, and Y. Berbers. A Design Process for Adaptive Behavior of Situated Agents. In *Volume 3382 of Lecture Notes in Computer Science*, 2005.

[10] D. Weyns, A. Omicini, and J. Odell. Environment, First-Order Abstraction in Multiagent Systems. *Journal of Autonomous Agents and Multiagent Systems*, 2006 (to appear).

[11] D. Weyns, K. Schelfthout, T. Holvoet, and T. Lefever. Decentralized control of E'GV transportation systems. In *4th AAMAS Conference, Industry Track*, Utrecht, 2005.

[12] D. Weyns, E. Steegmans, and T. Holvoet. Protocol-Based Communication for Situated Multiagent Systems. *3th AAMAS Conference, New York*, 2004.

[13] D. Weyns, E. Steegmans, and T. Holvoet. Towards Active Perception in Situated Multi-Agent Systems. *Journal of Applied Artificial Intelligence, 18(8-9)*, 2004.

[14] M. Wood, S. A. DeLoach, and C. Sparkman. Multiagent Systems Engineering. *Software Engineering and Knowledge Engineering*, 11(3), 2001.

[15] M. Wooldridge, N. Jennings, and D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, 3(3), 2000.

[16] F. Zambonelli and A. Omicini. Challenges and Research Directions in Agent-Oriented Software Engineering. *Journal of Autonomous Agents and Multiagent Systems*, 9(3), 2003.

[17] F. Zambonelli and V. Parunak. *From Design to Intention: Signs of a Revolution*. 1st AAMAS Conference, Bologna, 2002.