

Self-Adaptation Using Multiagent Systems

Danny Weyns and Michael Georgeff

*Each decade has its key software technology to advance artificial intelligence, and each technology is highlighted in a novel that sells much better than the underlying technology. Who hasn't read Michael Crichton's *Prey* and wondered how far multiagent systems might evolve and how they might affect humankind? Our technology column digs into this topic in this issue. Danny Weyns and Michael Georgeff provide a short introduction and show how multiagent systems help master the complexity of self-adaptive systems. They contrast multiagent systems with other current technologies and provide links and hints for practitioners who want to get started with this emerging field.*

I look forward to hearing from both readers and prospective column authors about this column and the technologies you want to know more about. —Christof Ebert

Modern distributed software systems such as Web-based e-commerce and business-process management pose huge engineering challenges. As the systems interact with one another, they necessarily become decentralized. The underlying system components and collaborations change over time—often in unan-

sible configurations and reconfigurations. Building systems that can handle unanticipated events is difficult and error prone. Self-adaptive systems call for a shift in engineering vision from satisfying offline requirements through traditional top-down methods to satisfying online requirements through the coordination of decentralized systems. This is where multiagent systems and agent-oriented software engineering can help.



anticipated ways. The systems must therefore make adaptations at runtime—that is, they need to be self-adaptive.

Engineering such systems requires concepts, methods, and infrastructures beyond what current practice offers. Conventional engineering approaches call for direct specification of all pos-

Self-Adaptive Systems

Self-adaptive systems respond dynamically to changes in their environment and user requirements. Self adaptation can apply to various system properties. For example, a self-healing system can automatically discover, diagnose, and correct faults; alternatively, a self-optimizing system can automatically monitor and adapt resource usage to ensure optimal functioning relative to defined requirements.

Current engineering practice takes an architecture-centric perspective on self-adaptive systems. Typical examples include the Rainbow framework developed at Carnegie Mellon University and IBM's blueprint for autonomic computing (see the sidebar for Web links). These approaches

provide an appropriate abstraction level for describing and managing dynamical system changes. Over the past decade, various frameworks have contributed to successful self-adaptive systems.

However, several challenges remain, including decentralized coordination of self-adaptation in a distributed setting. Decentralized control is crucial for quality requirements such as openness, robustness, and scalability. Global control of distributed systems such as Web-scale information systems, intelligent transportation systems, and power grids is difficult to achieve or even infeasible, although centralized control of local subsystems is possible.

Multiagent Systems

Multiagent systems belong to a class of decentralized systems in which each component (agent) is an autonomous problem solver, typically able to operate successfully in various dynamic and uncertain environments. These agents interact to solve problems that are beyond their individual capabilities or knowledge.

Multiagent systems have features that are key to engineering self-adaptive systems—specifically, loose coupling, context sensitivity, and robustness to failures and unexpected events. To some extent, loose coupling and context sensitivity are also present in conventional service-oriented systems. But self-adaptive multiagent systems extend these mechanisms to the behavior of each agent component.

Loose Coupling

Agents are self-contained, goal-directed entities. They get their adaptability from goals. When multiple agents are available, a goal can be achieved by selecting among the agents at runtime rather than requiring a hardwired design. For example, agents could negotiate a supply-chain collaboration at runtime on the basis of service providers' availability and specific preferences of the collaborating parties. This is similar to loosely coupled services in a service-oriented architecture (SOA) and yields the same flexibility and reuse benefits. However, because goals normally have a well-defined semantics (corresponding to some preferred world state or behavior), one agent can invoke others on

the basis of what they can achieve rather than their names. This gives agent-based systems more flexibility than service-oriented approaches.

Loose coupling and goal-directed behavior extend to an agent's internal processing, as typified by BDI (belief-desire-intention) architectures. For example, in a manufacturing execution system, an agent responsible for routing work pieces might adapt its routing strategy dynamically according to particular observations in its environment. By having a library of independent, semantically complete processes (rather than a hard-linked network of processes), a system can easily add new processes to the library, making it easy to extend the agent's capabilities. It's similarly easy to modify existing processes without having to rewire code in other processes.

Goal-based, loose coupling of agents externally and of agent processes internally provides the flexibility needed for self-adaptation and reuse. It also drives standardization and reduces total ownership costs.

Context Dependence

An agent includes a specification of the situation or context in which it's appropriate or expected to achieve its target goal. A calling agent can simply post the goals it wishes to achieve and select only those agents appropriate to the goal and current processing context: the right agent at the right time in the right circumstances. Similarly, an agent's internal processes are typically associated with a context con-

dition describing the situations in which the process can achieve its specified goal. This means that processes "self select" according to the desired goal and prevailing situation.

In conventional engineering, this contextual information isn't typically included in the definition of the called service or process (although service-oriented approaches move some way toward this objective). Instead—somewhat bizarrely when you think about it—the developer must write conditional decision logic in the calling process to ensure that the system will select the right called process. This complex decision logic rapidly leads to unmanageable code.

More troublesome, any changes to a process and the context in which it's appropriate require changes to the decision logic—not just in one place, but everywhere the process is used. As a result, developers can't create processes independently of one another.

Robustness

Goal-directed multiagent systems eliminate most of the complexity needed for handling agent or process failures, such as a delivery service's truck breaking down, and unanticipated events, such as a road closure. In such systems, failures and unexpected events cause the original goal to be reposted and tried again, without the need for explicit exception handling.

This way of handling failure is typical of the real world: if a door fails to open, try a key; if that also fails, ring the buzzer for someone to let you in. The goal is simple, but if we tried to write this in a conventional process language, we would have to explicitly specify all the exception-handling processes and how and when to apply them. In goal-directed agent systems, there is nothing to do. If other agents or processes can achieve the same objective, the goal-directed mechanism will automatically try them until success or ultimate failure.

Technology Comparison

Agent-based software engineering is often compared to object-based approaches. This might be useful for clarifying some aspects of agent technology, but the comparison is finally between apples and oranges. Agents are a specific architectural

Multiagent systems have features that are key to engineering self-adaptive systems—loose coupling, context sensitivity, and robustness.

style that imposes particular constraints on a system while yielding particular qualities and tradeoffs. Objects are the primary building blocks for implementing practical multiagent system architectures.

A more useful comparison contrasts agent-based approaches with more traditional architectural styles. Table 1 compares multiagent systems with client-server and service-oriented approaches in terms of usefulness, quality attributes, and cost.

A Self-Adaptive Automated Transportation System

An application for an automatic guided vehicle (AGV) transportation system illustrates the self-adaptive value of multiagent systems (for background information on this project, visit the Emc² Web site listed in the sidebar).

An AGV system transports loads in an industrial environment. Different clients can generate transports—for example, a warehouse management system, a machine's software, or a human operator. The system's main functionalities include assigning transport tasks to appropriate AGVs, routing the AGVs efficiently while avoiding collisions and deadlocks, and providing AGV maintenance services. The system must operate efficiently and

robustly while handling dynamic operating conditions such as AGVs leaving the system for maintenance, variable waiting times for production machines, delays in supply of goods, temporary closures in warehouse areas, and blocked paths.

Traditionally, a central server controls an AGV system. The server plans the schedule for the system as a whole, dispatches commands to the AGVs, and continually polls their status. This results in reliable, predictable solutions and easy diagnosis of errors. However, such solutions are usually inflexible and difficult to adapt to changing business needs. Systems that can autonomously adapt to changing circumstances must meet quality requirements for flexibility and openness. Specifically, they must have the flexibility to deal autonomously with dynamic operating conditions and the openness to deal autonomously with AGVs leaving and entering the system.

To meet these quality requirements for an AGV system, researchers from DistriNet Labs and engineers from EgeMin—the two partners in the project—developed a new architecture based on multiagent systems. Figure 1 shows a high-level deployment model of the system. An *AGV agent* deployed on a computer system in each vehicle controls the vehicle. A

transport agent deployed at a dedicated computer system called the *transport base* represents each transportation task in the system. AGV agents and transport agents must coordinate their actions—for example, assigning transports and avoiding collisions. Common middleware services manage communications over a wireless network.

To illustrate the system's self-adaptive properties, we explain how the agents use a dynamic communication protocol for transport assignment. The protocol extends the well-known contract-net protocol (CNET) that Reid G. Smith introduced in the early 1980s. DynCNET (Dynamic CNET) consists of five basic steps, shown in Figure 2.

1. The transport agent—the protocol's initiator—sends a call for proposals (cfp) to the AGV agents in its context—that is, the agents within a certain area from the load.
2. The AGV agents in this area—the participants in the protocol—respond with proposals.
3. The transport agent notifies the provisional winner.
4. While the AGV moves toward the load, the transport agent as well as the AGV agent can abort the provisional

Table 1

Technology comparison of multiagent systems with client-server and service-oriented system approaches

Description	Usefulness	Performance	Robustness	Adaptability	Scalability	Cost
Client-server	Centralized, reliable, high-security environments	Fast; response times increase gradually as more requests are made	System fails when server goes down; solve by increasing number of servers	Difficult to adapt to changing circumstances or new business requirements	Congestion risk when adding more users; solve by increasing number of servers	Higher initial capital investment, higher maintenance costs
Service-oriented	Loosely coupled systems with business- and technology-domain alignments	Dynamic service composition and description parsing introduce performance overhead	Depends on the quality attributes of the service composition infrastructure to deal with failures	Can adapt at the service level using dynamic service discovery	Stateful services require exchange of service (meta)data, which increases coupling and reduces scalability	Cost effective as a result of reusability, composability, and standardization
Multiagent system	Inherently distributed, locally autonomous, highly dynamic environments	Efficient adaptation to local changes; lack of global information can result in myopic decisions	Goal-directed mechanism handles failures; if failure occurs, it will have only local impact	Goal-directed, context-sensitive process selection extends the benefits of loose coupling and adaptability to individual processes and workflows	Depending on the connectivity, communication channel can become a bottleneck when adding nodes; solve by increasing bandwidth	Economical with regard to required processing power; poor integration of agent-based design tools with common engineering tools

agreement if a more suitable assignment is available.

- The selected AGV agent informs the transport agent when the AGV picks the load (bound).

The shaded zones in the activation boxes represent periods in the protocol when both agents can switch the provisional agreement.

The DynCNET protocol, in combination with the goal-directed, context-sensitive process selection, enables the agents to reconsider the environmental situation and adapt the task assignments dynamically when circumstances change. When a load is picked up or a new task enters the system, or when an AGV enters or leaves the system, the candidates for interaction will dynamically change and the agents will adapt their behavior accordingly. For example, an AGV that has provisionally accepted a certain task might decide, while moving toward its assigned load, to change the assignment to another load that now appears closer than the initially assigned load.

Tests in industrial installations have demonstrated up to 50 percent improvement in system throughput compared to the traditional static approach for task assignment. The trade-off is a doubling of required communications bandwidth.

Hints for Practitioners

Although multiagent systems are an appealing approach for developing decentralized self-adaptive systems, the gains don't come without cost. We report some important lessons learned from experiences with developing industrial-strength multiagent systems.

The Right Motivation

Quality requirements are the main drivers for structuring a software system. Multiagent systems are known for addressing quality attributes such as adaptability, robustness, openness, and scalability. The decision to apply a multiagent system architecture must be based on a good understanding of the stakeholders' main quality attributes and those realized by a multiagent system architecture.

Clarifying the added value and trade-offs of adopting a multiagent system will help architects make well-considered de-

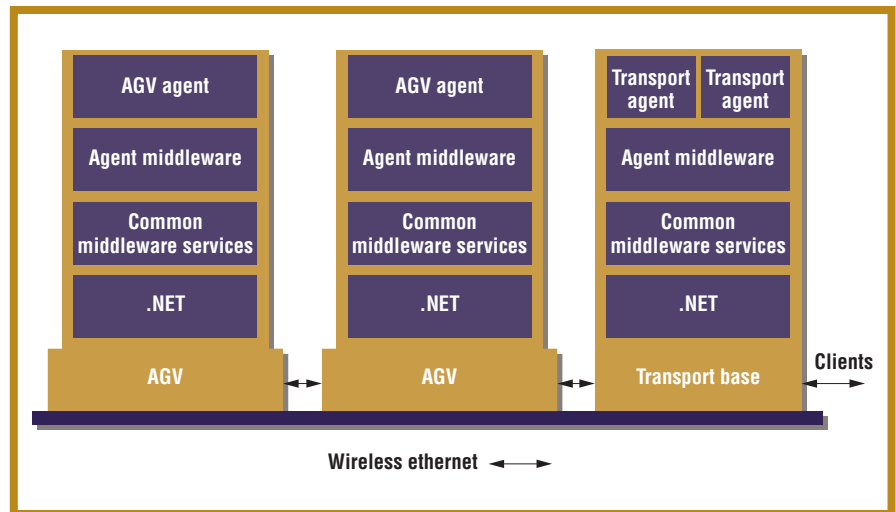


Figure 1. Deployment model of the automatic guided vehicle (AGV) transportation system. Each AGV is deployed with an AGV agent, and a transport base supports transport agents. Communications occur through a wireless Ethernet.

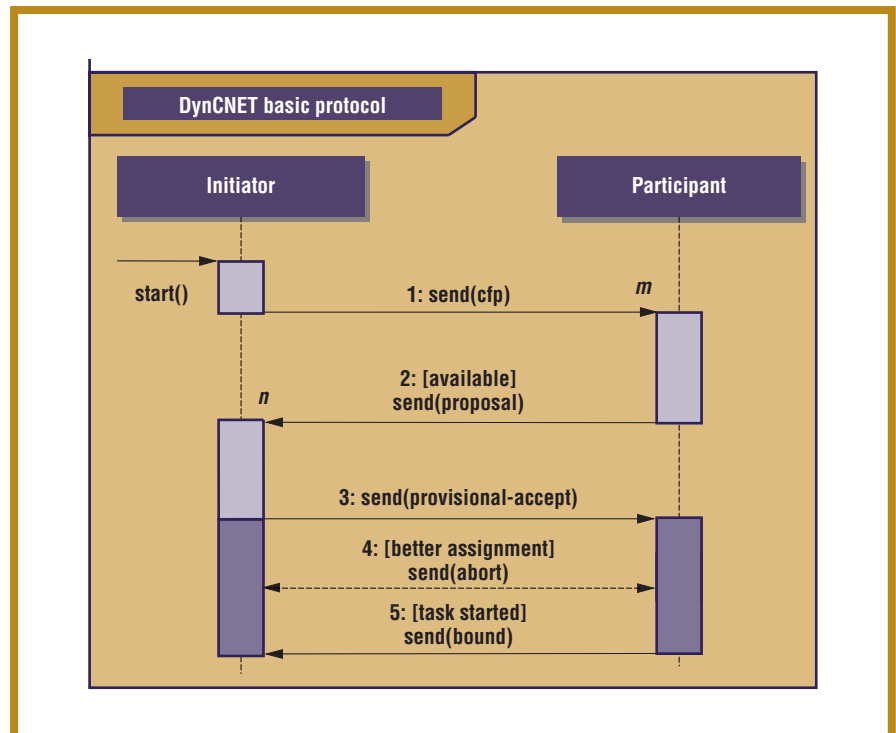


Figure 2. DynCNET protocol. The start() operation initiates the protocol. The transport agent—initiator—sends a call for proposals to m AGV agents—participants—within a certain area of the load. AGV agents respond with proposals to n transport agents within a certain area from their current location. After the provisional agreement, task assignment can be aborted, and the task can be reassigned until the load is picked up.

cisions and prevent stakeholders from overestimating or underestimating agent technology.

Multiagent System Integration

Software systems are rarely built in iso-

lation. Introducing a multiagent system usually requires embedding and integrating it with an existing software environment, including legacy systems. In multiagent system engineering, developers often consider “agentification” to be a general

Resources

Jeff Kramer and Jeff Magee provide an excellent overview of the state-of-the-art and challenges in self-adaptive systems in "Self-Managed Systems: An Architectural Challenge" (*Future of Software Engineering*, IEEE CS Press, 2007, pp. 259–268).

Anand S. Rao and Michael P. Georgeff introduce the basics of the belief-desire-intention agent architecture in "Modeling Rational Agents within a BDI Architecture" (*Proc. 2nd Int'l Conf. Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann, 1991, pp. 473–484).

In *Architecture-Based Design of Multi-Agent Systems* (ISBN 978-3-642-01063-7, Springer, 2009), Danny Weyns describes an architecture-based approach for multiagent system software engineering and its application to the AGV (automatic ground vehicle) transportation system that we describe in this article.

Additionally, Weyns and coeditors H. Van Dyke Parunak and Onn Shehory developed a special issue on the future of software engineering and multiagent systems for the *International Journal on Agent-Oriented Software Engineering* (vol. 3, no. 4, 2009, pp. 369–415). The issue bundles a set of papers that discuss the opportunities as well as the technical and organizational obstacles for industrial adoption of multiagent system technology.

Several Web sites offer information relevant to this article.

IBM Autonomic Computing takes a perspective on self-managing systems inspired by the human central nervous system; www.research.ibm.com/autonomic.

Rainbow is a framework for architecture-based adaptation of complex systems. It was developed at Carnegie Mellon University and supports automated, dynamic system adaptation via architectural models; www.cs.cmu.edu/~able/research/rainbow.

A Dagstuhl seminar dedicated to Software Engineering for Self-Adaptive Systems brought together researchers from different disciplines, including researchers with backgrounds in autonomic computing, dependable computing, robotics, multiagent systems, and service-oriented architecture; www.dagstuhl.de/de/programm/kalender/semhp/?semnr=08031.

Emc² (Egemin Modular Controls Concept) is a joint R&D project between Egemin and K.U. Leuven which has applied agent technology to developing a self-adaptive control system for an automated transportation system; <http://emc2.egemin.com>.

The Foundation for Intelligent Physical Agents is working with OMG on agent standardization, including a service-oriented architecture standard that supports agents (SoaML) and an agent metamodel and profile; <http://agent.omg.org>.

Living Systems of Whitestein Technologies is a pioneer and leading innovator in software agent technologies, autonomic computing, and self-adaptation; www.whitestein.com.

AdaptivEnterprise of Agentis Software is an innovative approach for developing adaptive software systems based on goal-directed agent theory; www.agentissoftware.com

The *International Journal of Agent-Oriented Software Engineering* aims to promote the interface between research and commercial adoption of agent technology and bring together agent technologists and conventional software engineers; www.inderscience.com/browse/index.php?journalCODE=ijaose.

with common middleware services. Notable exceptions are Whitestein Technologies' Living Systems and Agentis Software's AdaptivEnterprise platform, which are integrated with Java Enterprise Edition (see the sidebar for more information).

Multiagent System Design

The multiagent system community has developed a variety of agent-based methodologies. These methodologies have their value, but their specificity can hamper industrial adoption. Experience taught us that integrating agent-based techniques within mainstream software engineering works well for building practical multiagent systems.

For the AGV transportation system, we used the Software Engineering Institute's attribute-driven design method. To realize the system goals, we employed a set of multiagent system architectural patterns and combined them with some common architectural patterns. We organized an architectural evaluation using Carnegie Mellon's Architecture Trade-off Analysis Method to determine the design trade-offs and risks with respect to satisfying important quality attribute scenarios, particularly those related to flexibility, openness, and performance. The agent-based software architecture became a blueprint for system development, which we implemented in C#.

Multiagent System Testing

As with any distributed system, testing a multiagent system is challenging. Decentralization and deployment in open environments add to the complexity. Among the important issues to be considered when testing a multiagent system are dynamic interactions, nondeterminism, dependencies on third-party infrastructure, partial failures, semantic interoperability, task synchronization, and unwanted emergent behaviors.

You can combine traditional testing techniques such as unit and functional tests, advanced simulations, and partial formal verification to test a multiagent system.

Impact of Multiagent System Adoption

Conway's Law says that software architecture is related to a developing organization's structure. A dramatic change in the software architecture typically requires corresponding changes in the way teams

solution for integrating legacy code. However, concerns such as security, persistence, and transactional behavior often crosscut a system. Wrapping falls short when integrating existing infrastructure

that supports these concerns, which are typically provided as reusable middleware services.


Unfortunately, most available agent platforms do a poor job of integrating

are structured for developing, testing, and maintaining the software. Our experience indicates that moving from a traditional client-server architecture to a decentralized multiagent architecture is a big step with far-reaching effects not only for the software but also for the organization's structure.

One approach to manage this transition in a controlled way is to gradually shift responsibilities from the central server to the autonomous subsystems, focusing initially

on those tasks that benefit most from multiagent systems.

Complete offline design is no longer an option for distributed systems that must establish component collaborations at runtime and adapt dynamically with changing operational conditions and user needs. Multiagent systems can tackle some of the hard problems of engineering self-adaptive systems. Although they aren't a silver bullet, their added value will

be a critical advantage as software systems continue to integrate and decentralization becomes a matter of fact, and the added value of multiagent systems will be of critical advantage. 

Danny Weyns is a postdoctoral researcher at the Katholieke Universiteit Leuven's DistriNet Labs. Contact him at danny.weyns@cs.kuleuven.be.

Michael Georgeff is founder and chief executive officer of Precedence Health Care and professor in the Faculty of Medicine at Monash University. Contact him at michael.georgeff@precedencehealthcare.com.

CALL FOR ARTICLES

Software Architecture: Framing Stakeholders' Concerns

PUBLICATION: November/December 2010

SUBMISSION DEADLINE: 1 April 2010

When putting architecture viewpoints, frameworks, or models into practice, architects face recurring issues: Which views and models/languages do I need? How do I handle concern X? How do I illustrate the concerns addressed by my architecture to stakeholder Y? Are there any reusable viewpoints or models to frame the concerns of clients, auditors, or maintainers?

This special issue will explore the state of the art and current industrial practice in framing architectural concerns. We especially welcome case studies, lessons learned, success and failure stories in introducing viewpoints, frameworks, and models to organizations, mature and innovative approaches, and future trends.

POSSIBLE TOPICS INCLUDE BUT ARE NOT LIMITED TO

- research approaches and industrial practice on identifying, documenting, and applying viewpoints, frameworks, and models (VFMs) in framing architectural concerns;
- tools to support VFMs in framing architectural concerns;

- reuse, customization, generalization, and standardization of architectural VFMs;
- viewpoints and models for specialized concerns (e.g., reliability, safety, security) or for specific domains (enterprise, healthcare, embedded systems); and
- relations between VFMs with other knowledge management mechanisms such as perspectives, principles, styles, and patterns.

QUESTIONS?

For more information about the focus, contact the Guest Editors:

- Patricia Lago, VU University Amsterdam, patricia@cs.vu.nl
- Paris Avgeriou, University of Groningen, paris@cs.rug.nl
- Rich Hilliard, software systems architect, r.hilliard@computer.org

For author guidelines:

www.computer.org/software/author.htm

For submission details: software@computer.org

IEEE
Software