An Agent Design Method Promoting Separation Between Computation and Coordination

Nico Janssens, Elke Steegmans, Tom Holvoet, Pierre Verbaeten
DistriNet, Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A
B-3001 Leuven, Belgium
{nico.janssens, elke.steegmans, tom.holvoet, pierre.verbaeten}@cs.kuleuven.ac.be

ABSTRACT

The development of (internet) agents is often a tedious and error-prone task resulting in poorly reusable designs, since both the *internal computation* of the agent as well as the *coordination support* are developed in an *ad hoc* fashion. To improve the process of agent-oriented software development, we propose an *agent design method* that imposes the separation of internal computation from coordination aspects. This method comprises two dimensions: a *design formalism* and an *agent design process*. As an illustration of the presented method, we present the design of an internet agent that is entitled to deploy a distributed service in a computer network, without breaking the consistency of that network. The presented design method has resulted in the development of ACF (Agent Composition Framework), a component framework to build flexible internet agents. We argue that the presented design method combined with this infrastructure can promote a modular and easy to manage approach to the design and development of internet agent applications.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*agent design method*

Keywords

agent design method, coordination, separation of concerns, case-study

1. INTRODUCTION

Over the past few years, the Internet has been evolving rapidly along a number of dimensions. This can be perceived by an obvious trend towards large-scale networks with a vast quantity of available information. In addition, these networks are made up of a multiplicity of heterogeneous components ranging from servers to mobile phones, as such accommodating a various collection of services. For the past decade, agent technology has proven to be very suitable (over other approaches such as the client/server paradigm) to fully exploit the potential of this open, distributed, heterogeneous, decentralized and unpredictable nature of the 'Internet of the future'. Examples of agent applications in the Internet range from network management [2] to web-based applications [20].

Due to the distributed environment these internet agents operate in, *coordination* is a fundamental aspect of the agent behavior in order to accomplish their task. An agent must be able to *communicate* and *cooperate* with other agents in order to exchange information or ask their help in pursuing a goal. However, the term 'coordination' has become an umbrella for a broad area of research activity. In order to put the rest of this work into some perspective, we focus in the remainder of this paper on *basic coordination infrastructure*, as defined in [18]: "The models in this category do not deal with coordination mechanisms *per se*, but instead they provide the means necessary to build fully fledged coordination frameworks."

The development of (internet) agents is often a tedious and error-prone task, resulting in poorly reusable designs since both the *internal computation*[1] of the agent as well as the *coordination support* are developed in an *ad hoc* fashion. To improve the process of agent-oriented software development, there has already been done a lot of work in the area of agent-oriented methodologies. Well-known examples are AUML [16], Gaia [23], MESSAGE [5] and Tropos [9]. However, these methodologies focus on the general behavior of software agents, without providing separate abstractions for defining the employed coordination model. As a consequence, reuse and changes with respect to the employed coordination model are often very impractical. To make the design process of internet agents easy, accessible and reusable, we propose an *agent design method* that imposes the separation of internal computation from coordination aspects. This method has resulted in the development of ACF (Agent Composition Framework), a component framework to build flexible internet agents. We argue that the presented method combined with this infrastructure can promote a modular and easy to manage approach to the design and development of internet agent applications.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC '04, March 14-17, 2004, Nicosia, Cyprus
Copyright 2004 ACM 1-58113-812-1/03/04 ...$5.00.

[1] the term *internal computation* is used to define the behavior of an agent, separated from communication concerns

The remainder of this paper is structured as follows. Section 2 describes the agent design method we propose, and demonstrates how the functionality (representing the internal computations) of a internet agent can be separated from the employed coordination model. As an illustration of the presented method, we describe in section 3 the design of an internet agent that is entitled to deploy a distributed service in a computer network, without breaking the consistency of that network. Next, the abstractions of the resulting ACF framework are outlined in section 4. In section 5, some related work is discussed. Finally, the paper ends with a conclusion and future work in section 6.

2. AGENT DESIGN METHOD

The agent design method we present comprises two dimensions.

First, there is a need for a *design formalism* that clearly separates the internal computation of an agent from the imposed coordination model. To achieve this, we have validated the formalism to model an agent's behavior as described in [11]. This formalism turned out to be a very useful basis for defining coordination related extensions. As such, we propose some extensions to separate away the employed coordination model from the regular behavior of an agent. This formalism as well as the coordination related extensions are described in section 2.1.

Second, using this formalism we propose an *agent design process* consisting of two cycles. In a first cycle the internal computation of an agent is defined, making abstraction of coordination specific issues. Next, a second cycle comprises the orthogonal extension of the basic model with the employed coordination model. This will be elaborated on in section 2.2.

2.1 Formalism to Model Agent Behavior

The formalism to model the behavior of an agent as described in [11] is based on statecharts. Such a statechart is used to define the *role* an agent represents in an application; it describes the *internal computation* of the agent. The primitives of these statecharts (illustrated in figure 2) are based on the concepts of a Mealy statechart [14]:

- a *state* describes the execution state of the agent at a certain moment in time. Such a state is represented by an oval.
- a *transition* connects two states with each other. A transition is depicted by means of a directed arrow between two states. 2
- an *action* (which is added to a transition), defines the actual behavior of an agent: it models the functionality that should be performed successfully by an agent to move over from an old state into a new state. An action is depicted by a white rectangle attached to a transition.

In addition to these basic primitives, the formalism presents five concepts (preconditions for actions, pre-actions and post-actions, pre-state-actions and post-state-actions) to extend the internal computation of an agent in a *modular* manner [11]. These concepts are a means to 'enter' the role of an agent, and to extend this model with additional cross-cutting concerns. Stated differently, they allow a designer to define orthogonal concerns that cross-cut the internal agent computation in a modular way. Therefore they are an interesting means to extend the model of that behavior with coordination aspects. Rather than elaborating on all of these concepts, we shortly describe those that turned out to be relevant for adding coordination aspects to the basic agent behavior.

- a *pre-action*, defines the functionality that should be completed before an action is executed. Concatenation of a number of pre-actions of an action is allowed. A pre-action is depicted by a light gray rectangle placed in front of an action.
- a *post-action* defines the functionality that should be completed after the action is executed. Again, it is possible to concatenate a number of post-actions. A post-action is depicted by a light gray rectangle located after an action.

With a view to separate the coordination aspects of an agent away from its internal computation, we have defined coordination specific variants of the pre-actions and post-actions, labelled resp. *coordination pre-actions* and *coordination post-actions*. These concepts are used exclusively for modelling the coordination aspects of an agent.

2.2 Agent Design Process

The separation of concerns with respect to coordination and internal computation entails that adding, removing or changing the coordination mechanism of the agent does not require any reprogramming of the agents themselves; the inter-agent coordination mechanisms act as *wrappers* around the actions (defining the internal computation) of the agent.

As such, in order to make the development of internet agents less tedious and error-prone, we propose an agent design process consisting of two cycles. Each cycle represents a different development role. In a first iteration, called the *internal computation design cycle*, the internal computation of the agents is modelled using the *state*, *transition* and *action* primitives as described in the previous section. In this role, an agent developer abstracts away all coordination related details. In a next iteration, called the *coordination design cycle*, the designer concentrates on the orthogonal extension of the basic model with coordination aspects. This is accomplished by 'weaving' *coordination pre-actions* and *coordination post-actions* into the model that represents the internal agent computations. In this phase, decisions are taken with respect to 'where' in the basic model coordination support is required and 'what' technology will be used to implement it. This design process allows concentrating on a single concern without being distracted by other concerns scattered across the same functional code.

3. CASE STUDY

As a proof of concept for the agent development method presented in the previous section, we discuss an example that illustrates the development process of a *Distributed Software Deployment Agent*. Before elaborating on this process, we shortly describe the problem domain of the example in section 3.1. Next, we focus on the agent design in section 3.2.

3.1 Problem Analysis

Currently, the use of *adaptive software* (that is, software that can be changed at runtime) increases significantly in the

area of *critical distributed systems*, ranging from computer networks over middleware to application servers. The benefits of runtime software adaptations vary from the ability to respond rapidly to security threats to the opportunity to optimize performance. Unfortunately, implementing adaptive distributed software is also more challenging in such systems for a variety of reasons, including the need to *coordinate adaptations* among multiple machines [7].

The increasing demand for *adaptive computer networks* has resulted in the rise of the programmable networks paradigm [6]. These networks differ from regular computer networks in their ability to become programmed. In contradiction to legacy routers, programmable routers are open and can be customized, e.g. by deploying specific and non-standardized communication protocols. However, most communication protocols (such as for instance compression, encryption or fragmentation services) consist of *collaborating entities* each located at different routers or hosts, rather than relating to one single router. In addition, these distributed communication protocols often employ a *bidirectional communication model*. This implies that both upstream and downstream communication between two routers are subject to the communication protocol that has been deployed.

In order to demonstrate the need for *coordinated adaptations* to realize the correct runtime deployment of such distributed, bidirectional communication protocols, we discuss the *upgrade of an encryption protocol*. The applied network setup is illustrated in figure 1. Both routers are equipped with an encryption and a decryption module, each responsible for resp. encrypting the data packets that enter the secure communication channel and decrypting the ones that leave it. In order to achieve the runtime replacement of an old encryption protocol by a more safer variant, each router has been equipped with a *Distributed Software Deployment Agent* (DSD Agent). This agent is responsible for replacing both the encryption and decryption module of the visited router with the newer version. However, when the distributed dependencies of the encryption modules towards their peer entities (located at the other host) are ignored during the upgrade process, the correct functioning of the encryption services will get disrupted. When the DSD Agent of router 1 replaces the encryption module before an appropriate decryption module has been installed by the DSD Agent of router 2, packets encrypted by the new algorithm will be discarded.

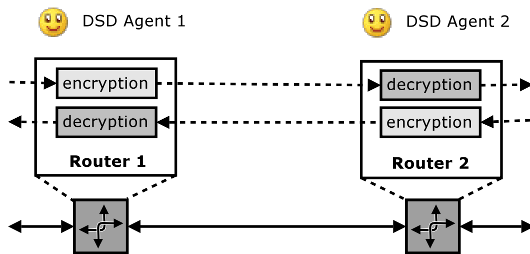

Figure 1: Simplified network setup consisting of two routers connected with a secure communication channel

Therefore, to preserve the integrity of the network while the adaptation is in progress, a coordination mechanism is required that involves doing the service deployment in a coordinated manner. This type of coordination mechanism has been entitled *inter-host coordination* [7], and should be added to the DSD Agent in order to work correctly.

3.2 Agent Development

3.2.1 Internal Computation Design Cycle

During the *internal computation design cycle*, the internal computations of the DSD Agent have been designed. Before elaborating on the model that reflects the internal computation of the DSD Agent, we shortly outline the employed process to achieve graceful runtime replacement of a (uni-directional) encryption protocol. This process comprises three phases: *deployment of the new protocol*, *activation of the new protocol* and finally the *removal of the old protocol*.

Deployment of the new protocol. Initially, the procedure starts by the deployment of the new encryption protocol, resulting in the *co-existence* of the old protocol (still in use), and the new version (not yet activated). This is achieved by first adding the new decryption module to router 2, followed by the deployment of the new corresponding encryption module on router 1. Since both the old and the new protocol will execute in parallel during the reconfiguration process, additional support to mark packets that are handled by the new encryption protocol (in order to distinguish them from packets processes by the old protocol) is inserted at router 1 as well.

Activation of the new protocol. Next, the newly deployed encryption protocol becomes activated. This is achieved by stopping the old encryption module, and activating the new version. As such, packets that are passing by will now be processed by the new encryption service.

Removal of the old protocol. Finally, the old protocol should become removed. Removing the old encryption module is trivial, since it has been deactivated in the previous phase. However, there might still be packets on their way that have been processed by the old encryption module. Therefore, the old decryption module (located at router 2) can only be removed when all packets processed by the corresponding encryption module are correctly received. When this 'finished' state is reached, the original decryption module also becomes removed. Finally, since at this point all packets in between both routers are processed by the new encryption protocol, the marking support at router 1 becomes redundant and will get removed.

As stated above, the DSD Agent is responsible for managing the deployment of *bi-directional* protocols. As such, each agent controls the replacement of both the *sending part of the downstream protocol* (in this example the encryption module) as well as the *receiving part of the upstream protocol* (the decryption module). Taking this into account, the design model of the DSD Agent reflecting its *internal computations* is illustrated by the states, transitions and actions as depicted in figure 2.

3.2.2 Coordination Design Cycle

In a next iteration, described as the *coordination design cycle*, the model of the DSD Agent has been extended with coordination aspects. As stated above, coordination between the DSD Agents located at routers 1 and 2 is essential to realize the correct runtime deployment of the new encryption protocol. Therefore, decisions are taken with respect to

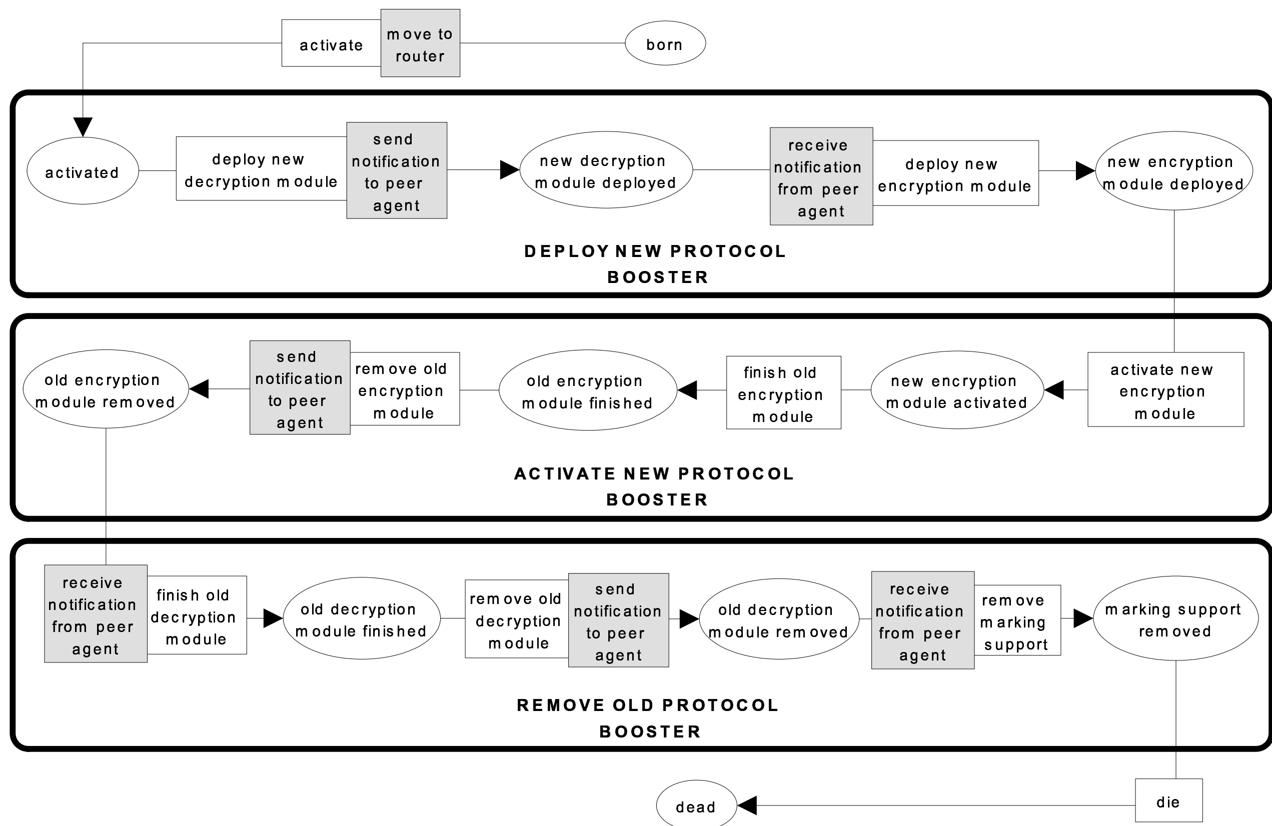

Figure 2: Design model of the Distributed Software Deployment Agent

where coordination aspects should be 'woven' in the model. This has resulted in the injection of a number of *coordination pre-* and *post- actions*. Coordination pre-actions on the one hand, are used to introduce the *synchronization points* in the model where the DSD Agent expects notification from its colleague before continuing its execution. This has resulted in the addition of a number of `receive notification from peer agent` coordination pre-actions as illustrated in figure 2. Coordination post-actions on the other hand, are added to *define communication* towards the other agent, as such providing in the notifications the other agent is expecting. This has resulted in the addition of a number of `send notification to peer agent` coordination post-actions as depicted in figure 2. Note that none of both extensions has affected the internal computation of the DSD Agent.

Next to defining 'where' in the basic model coordination support is required, the agent designer also defines *what technology* will be used. For this example, both *direct communication technologies* could be chosen (e.g. by using RMI [1]) as well as *indirect communication technologies* that make use of the environment (e.g. by using tuple spaces based technologies like LIME [19]). Selecting a different communication technology does not affect the internal computation of the agent and vice versa.

4. FRAMEWORK

The design method as presented in the previous sections provides an instrument to improve the agent-development process. However, there is still a gap between the design and implementation of an internet agent. In order to bridge that gap, a component framework for implementing flexible agents starting from the models presented in section 2 has been developed. This framework has been named ACF (Agent Composition Framework) and has been derived from DiPS [15], a component framework to build adaptable protocol stacks. Frameworks like ADK [10] and JGram [21] have proven that component technology is a promising means to implement agent logic. We argue that the abstractions offered by the ACF infrastructure can promote a modular and easy to manage implementation of the coordination model applied to an agent application.

4.1 Technology background

ACF brings together three important research tracks in software engineering research and practice : a *plug-compatible component model*, *pipe-and-filter architectures* and *separation of concerns*:

- As a method for creating maintainable and customizable software, a **pipe-and-filter** architectural style [8] has been proposed. Such an architecture forces an (agent) developer to define components that contain pieces of (agent) behavior. These components are plugged one after the other to create a functional system. The ACF has adopted this method in order to implement flexible and maintainable agent roles.

- Component frameworks are often referred to as black box frameworks that accept **plug-compatible components** [22]. In order to increase flexibility, ACF components are kept independent from each other by employing a *plug-compatible component model*. This is a major advantage concerning flexibility, because it allows individual components to be *reused* in different compositions, representing different agent roles.
- **Separation of concerns**. Strict separation of non-functional behavior from the functional code has proven to be an essential feature for building adaptable, maintainable and reusable software [12]. Therefore, in order to allow customization of the used coordination model of an agent without affecting its basic behavior and vice versa, ACF aims for separation of the coordination model from the functional model.

4.2 Component Framework Abstractions

The research tracks as discussed in the previous section have resulted in the design of *ACF components* as first class entities. The employed plug-compatible component model has resulted in the development of ACF components that are equipped with predefined *incoming* and *outgoing ports*. Communication between two neighboring components is achieved by means of *connectors*, which are responsible for connecting their incoming and outgoing ports to each other. As such, ACF components are kept *unaware* of the counterparts to which they got attached, which improves reuse of ACF components in different compositions.

Semantically, the ACF components are a first class representation of the concepts defined in section 2, imposing a number of restrictions to the agent programmer. We explain these components and their restrictions shortly.

- **ActionComponent**. This component has one incoming and one outgoing port. The content of the component must be restricted to internal computation functionality of an agent.
- **PreActionComponent** and **PostActionComponent**. These components also have one single incoming and one outgoing port. These components must only be implemented with orthogonal concerns that cross-cut the internal agent computation. In order to extend the internal computation of an agent with coordination support, **CoordinationPreActionComponents** and **CoordinationPostActionComponents** are offered.

In addition, tool support is currently being developed to translate the design model of an agent into a valid first class representation using ACF components, as such bridging the gap between the design and implementation of an internet agent.

5. RELATED WORK

In order to improve the development of agents and (complex) multi-agent systems, it is important to have a methodology that supports the analysis and design of such a (multi-)agent system. Focussing on how coordination is handled by other agent development methodologies, we conclude that most of them only deal with interactions occurring *directly between agents*, mostly ignoring *interactions of the agents with the environment*. Therefore, one of the advantages and aims of our design method is to support both kinds of interactions, in such a way that the type of interaction is added as a separate concern to the behavior of the agents. In the remainder of this section we elaborate on a number of existing methodologies.

AUML [16] provides support to model the autonomous behavior of agents and their interactions. Statecharts and activity diagrams are offered as the primitives to model these interaction protocols. However, AUML only supports modelling *direct interactions* between agents. As such, it is not possible to model interactions of the agents with the environment (e.g. by using tuple space based technologies). Using our design method both direct and indirect agent interactions can be modelled (as indicated in section 3.2.2). The new concepts added to our statechart formalism can be used to extend the behavior of agents with coordination aspects. The major advantage of these concepts is the clear separation between the computation and the coordination of the agents. Using AUML, agent interactions are added to the statecharts by means of actions and it is not clear which are the *interaction* actions and which present the *computation* of the agent. The only notion added to differ between computation and coordination is a *comment* added to the "interaction action", depicted by a dotted arrow and indicating that it is an interaction with another role.

In [24] the GAIA [23] methodology has been extended to support coordination models and as such to make GAIA suitable for the development of Internet-based applications. Although Zambonelli et al [24] have made GAIA coordination-oriented, they conclude that their coordination-oriented GAIA methodology is far from well-defined yet. It requires a more detailed specification supported by an appropriate formalism, which is one of the main contributions of our work: the extended statechart formalism is an appropriate formalism to add other concerns such as for example coordination to the behavior of the agents.

TuCSoN [17] and MARS [4] are two well known coordination architectures, based on the definition of Linda-like tuple spaces, providing coordination between agents among themselves and between agents and the environment. Both of them promote and provide a clear separation between computation and coordination. The proposed design process in this paper also focuses on the separation between computation and coordination, but more important and different from MARS and TuCSoN is our focus on defining the needed coordination of the agents *independent* from the chosen coordination medium and rules. As such, generic coordination aspects can be defined and added separately to the computation aspects of the agents. MARS and TuCSoN on the contrary focus only on coordination aspects depending on their specific Linda-based coordination medium.

6. CONCLUSIONS AND FUTURE WORK

An important characteristic of our design method and in particular of our extended statechart formalism is the focus on *separation of concerns*. The advantages that are brought along are twofold. First, the primitives of the presented formalism offer great advantages in expressive comfort. One can easily determine which part of an agent model defines the coordination aspects. Second, easy adaptations of an agent's behavior is allowed, as well as reuse of (parts of) the modelled behavior of an agent in different applications. Changing the coordination model of an internet agent does

not affect its internal computation and vice versa.

Quoting [3] “An interesting issue from the viewpoint of agent-oriented programming is that the coordination of the agent should not require any reprogramming of the agents themselves.”, we conclude that the design method proposed in this paper is a possible solution for this research topic in case of applications comparable to the DSD Agent. A key question that remains is whether this design method is applicable to a broad range of agent applications. As such, future work involves validation of the presented design method on other real life cases.

In addition, future work involves following topics. The use of Aspect Oriented Programming [13] will be investigated to separate internal computation from the employed coordination model. Finally, the tool for converting an agent model into ACF components will be further developed.

7. ACKNOWLEDGMENTS

This paper presents results from research sponsored by the research council of the K.U. Leuven. The results have been obtained in the *Concerted Research Action on Agents for Coordination and Control – $AgCo_2$ project.*

8. REFERENCES

[1] Java Remote Method Invocation (RMI). http://java.sun.com/products/jdk/rmi/.

[2] A. Bieszczad, T. White, and B. Pagurek. Mobile Agents for Network Management. *IEEE Communications Surveys*, 1998.

[3] N. Busi, P. Ciancarini, R. Gorrieri, and G. Zavattaro. Coordination Models: A Guided Tour. In *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 6–24. 2000.

[4] G. Cabri, L. Leonardi, and F. Zambonelli. MARS: a Programmable Coordination Architecture for Mobile Agents. IEEE Internet Computing, Vol. 4, No. 4, pp. 26-35, July-August 2000., 2000.

[5] G. Caire et al. Agent Oriented Analysis Using Message/UML. In *AOSE*, pages 119–135, 2001.

[6] A. T. Campbell et al. A Survey of Programmable Networks. *ACM SIGCOMM Computer Communications Review*, 29(2):7–23, April 1999.

[7] W.-K. Chen, M. Hiltunen, and R. Schlichting. Constructing Adaptive Software in Distributed Systems. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 635–643, Phoenix, AZ, April 2001.

[8] D. Garlan and M. Shaw. An Introduction to Software Architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume 1. 1993.

[9] F. Giunchiglia, J. Mylopoulos, and A. Perini. The Tropos Software Development Methodology: Processes, Models and Diagrams. In Proceedings of the Third International Workshop on Agent-Oriented Software Engineering III - AOSE 2002 , Bologna, Italy 2002, 2002.

[10] T. Gschwind, M. Ferudin, and S. Pleisch. ADK: Building Mobile Agents for Network and System Management from Reusable Components. In *Proceedings of Symposium on Agent Systems and Applications / Symposium on Mobile Agents (ASA/MA'99)*, pages 13–21, 1999.

[11] T. Holvoet and E. Steegmans. Application-Specific Reuse of Agent Roles. Software Engineering for Large-Scale Multi-Agent Systems, Lecture Notes in Computer Science, April, 2003., 2003.

[12] G. Kiczales. Towards a New Model of Abstraction in the Engineering of Software. In *Proceedings International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, Tokyo, Nov. 1992.

[13] G. Kiczales et al. Aspect-Oriented Programming. In *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP97)*, pages 220–242. 1997.

[14] G. H. Mealy. A method for synthesizing sequential circuits. Bell System Technical Journal, vol. 35, no. 5, pp. 1045–1079, 1955., 1955.

[15] S. Michiels. *Component Framework Technology for Adaptable and Manageable Protocol Stacks*. PhD thesis, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, Nov. 2003.

[16] J. Odell, H. Parunak, and B. Bauer. Extending UML for Agents. In Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence., 2000.

[17] A. Omicini and F. Zambonelli. Tuple Centres for the Coordination of Internet Agents. In proceedings of the 1999 ACM Symposium on Applied Computing (SAC '99) Special Track on Coordination Models, Languages and Applications., 1999.

[18] G. Papadopoulos. Models and Technologies for the Coordination of Internet Agents: A Survey. In *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 25–56. 2000.

[19] G. P. Picco, A. L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In *International Conference on Software Engineering*, pages 368–377, 1999.

[20] J. Shavlik and T. Eliassi-Rad. Intelligent agents for Web-based tasks: An advice-taking approach. In *AAAI/ICML Workshop on Learning for Text Categorization*, 1998.

[21] R. Sukthankar, A. Brusseau, R. Pelletier, and R. Stockton. JGram: Rapid Development of Multi-Agent Pipelines. In *First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents ASA'99 and MA'99*, pages 30–40, Palm Springs, California, USA, 3-6 Oct 1999.

[22] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. 1998. ISBN 0-201-17888-5.

[23] M. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, pages 285–312, 2000.

[24] F. Zambonelli et al. Agent-Oriented Software Engineering for Internet Applications. In *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 326–346. 2000.