2009 Fifth International Conference on Autonomic and Autonomous Systems

An Observation Framework for Multi-Agent Systems

Joonas Kesäniemi, Artem Katasonov and Vagan Terziyan
University of Jyväskylä, Finland
joonas.kesaniemi@jyu.fi, artem.katasonov@jyu.fi, vagan@jyu.fi

Abstract

Existing middleware platforms for multi-agent systems (MAS) do not provide general support for observation. On the other hand, observation is considered to be an important mechanism needed for realizing effective and efficient coordination of agents. This paper describes a framework called Agent Observable Environment (AOE) for observation-based interaction in MAS. The framework provides 1) possibility to model MAS components with RDF-based observable soft-bodies, 2) support for both query and publish/subscribe style ontology-driven observation, and 3) ability to restrict the visibility of observable information using observation rules. Additionally, we report on an implementation of the framework for the JADE middleware platform, where AOE is realized as a custom kernel service.

1 Introduction

A classic definition of an *agent* is an encapsulated computer system situated in some environment and capable of flexible, autonomous action in that environment in order to meet its design objectives [25]. The property of being situated in an *environment* was traditionally seen as one of the basic characteristics of an agent. The agents are described as entities that have *sensors* to perceive the state of the environment as well as *actuators* to affect the environment. In a multi-agent system, from the point of view of an individual, all the other agents in the system are a part of the environment. If a distinction is to be made, this environment is *social* in contrast to the physical one. The ability to interact with the social environment is as important as the ability to interact with the physical one. It is also the cornerstone of *coordination* of agents which is one of the fundamental problems in multi-agent systems [18].

Because a social environment consists of agents, i.e. active entities, there are several ways in which an agent can interact with its social environment:

- Communication – exchanging messages with other agents.
- Direct control – affecting, without communication, properties or available options of other agents, e.g., killing an agent or blocking an agent's way.
- Indirect observation of actions – observing the changes in the physical environment that occur due to actions taken by other agents, e.g., a door became open.
- Direct observation of actions – observing an agent performing an action, e.g., an agent opening a door now.
- Observation of properties – directly observing a bodily property of an agent, e.g., physical dimensions or what sensors and actuators it has.

In the classic approach to implementing multi-agent systems, realized in frameworks such as AgentSpeak with corresponding platform Jason [16, 3] and 3APL [6], the environment is *explicitly* represented. The environment is programmed as an object separate from the agents. Moreover, the agents do not possess sensors and actuators, rather the environment provides those. This means that the agents sense or act by invoking some methods provided by the environment's implementation. For example, if the chess game is to be implemented as a multi-agent system, the agents would be the chess pieces and the environment would be the chess board. The board would then provide methods enabling a piece to move or to check the positions of other pieces.

With such an implementation, the four first types of interactions in the list above are easy to realize. Since all sensing and acting is done through the environment object, this object naturally has a nearly complete view on the state of the multi-agent system. It is of no problem to realize, e.g., the direct observation of actions. The environment can be instructed to notify all the agents that some action is taking place now, e.g., a chess piece is moving. The observation of agent's properties is the only type of interaction that is not immediately enabled since those properties are intrinsic to agents, and information on them does not go through the environment object.

In recent years, the focus has shifted towards better support for *distributed* multi-agent systems. IEEE FIPA has

978-0-7695-3584-5/09 $25.00 © 2009 IEEE
DOI 10.1109/ICAS.2009.55

standardized the FIPA abstract architecture [8] according to which the agents are fully independent entities only supported by a *middleware* platform. Such middleware has to provide a set of application- and domain-independent services such as life-cycle support, message transport, and discovery mechanisms. In this architecture, the environment is *implicit*. The middleware itself is a shared part of the environment, but its responsibilities are mainly restricted to enabling communication between agents. As agents can be distributed over a network, each of them can have its own environment and thus own sensors and actuators. If some agents do share an environment, the middleware is not responsible for providing any assistance with that. Most recent frameworks for implementing multi-agent systems either directly implement FIPA specifications, such as JADE [1], or have own architecture yet following FIPA's basic middleware-based approach as in the cases of, e.g., Cougaar [9] and AgentFactory [5].

Although such middleware-based architecture provides important advantages with respect to autonomic and distributed operation, an inherent problem is that not only the observation of agents' properties but also the direct observation of actions is not supported any longer. Consider, for example, a hostile agent that enters a multi-agent system with malicious intentions. In the world of humans, the others could notice a suspicious "configuration" of the hostile person (e.g. he carries a gun) or some suspicious actions that he takes. In a middleware-based multi-agent system, none of those is possible unless the agent decides to communicate those facts (which a hostile agent most likely will not do). Therefore, there is a call for a mechanism, as a part of the middleware platform, with predictable and configurable behavior that would be in charge of managing each agent's observable state and observable behavior.

Observation can be associated with many different types of systems as shown in [19]. Our goal is to make observation a first class entity in MAS. The approach was inspired by the work of Weyns et al. [23] and their efforts towards explicit modeling of the environment. Providing observability is identified as one of the main responsibilities of the environment [22]. Starting from that premise, we have come up with an idea of providing agents in MAS with an observation mechanism similar to our physical world. The visual observations, for example, are transferred through shared medium and received using identical instrument (the human eye). The visual stimulus caused by the physical artifact is not intentionally created, rather than result of the nature of the medium. The presence of a human, for example, is always observable, but the observation made about the clothes he or she is wearing, are result of an intentional actions made earlier by the source of the observations. Our framework aims to provide a similar medium for observation for the virtual agents.

In this paper, we present a framework called Agent Observable Environment (AOE). AOE integrates information from various source into one shared, observable state of the world. In addition to the intentional manipulation of the observable state by the agent itself, the framework also support specification of environment based observable information that is superimposed on an agent (environment's intention). Agent can intentionally "show" some things, but there can also be things that it cannot "hide". The observable state is then accessible to all observers for querying and publish/subscribe style access. The framework is application- and domain-independent and can be used as a basis for creating observation based coordination mechanisms for MAS.

The rest of the paper is structured as follows. Section 2 describes the Agent Observable Environment framework. Section 3 reports on our AOE implementation for the JADE middleware platform. Section 4 comments on the related work, and, finally, Section 5 concludes the paper.

2 Agent Observable Environment

The Agent Observable Environment (AOE) is a framework that models the observation process in multi-agent systems (MAS) as interaction between the observer and its local observable environment (OE). The local OE, consisting of a set of soft-bodies, is governed by the observation rules of the system. The term soft-body was originally defined as an explicit boundary between agent's internal machinery and the environment the agent inhabits [15]. For the purposes of AOE, we extend the realm of entities capable of having a soft-body to include also MAS services that provide support for mediated agent interaction, such as digital pheromone infrastructure [14]. In AOE, the soft-body works as a container for entity-specific observable information. It is modeled as a part of the entity, but managed by an environmental service. The global observable environment, which can spawn all the nodes of a distributed agent platform, is then created as the sum of all the soft-bodies in the system.

Figure 1 illustrates the logical architecture of the AOE using the layered representation of MAS presented in [20]. The arrows from agents and services to the observable environment represent the changes made to their observable states; i.e. their soft-bodies. The dotted arrows in the opposite direction represent the observations that can contain either information about the current (static) state of the observable environment or its (dynamic) evolution. So, the main environment abstractions provided by the AOE are soft-bodies, manifestations of observations and observation rules.

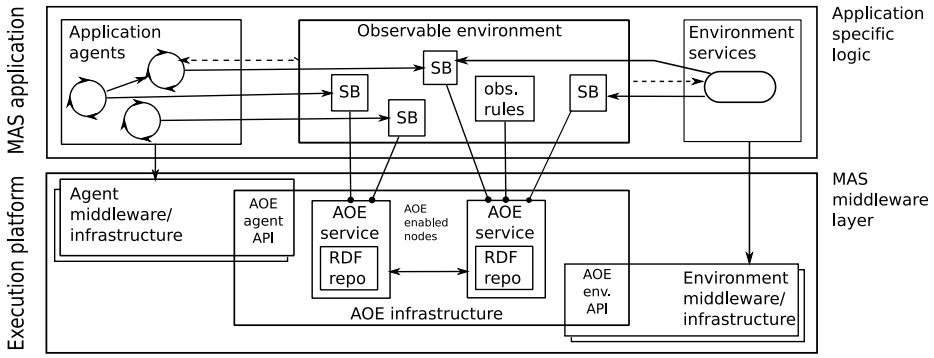

Figure 1. AOE logical architecture

2.1 Soft-body

As mentioned earlier, the soft-body is the repository for the observable state of an entity. The soft-body data is modeled using Resource Description Framework (RDF). RDF was chosen as the data model because of its flexibility, implementation independency and support for ontology-based reasoning. The first two features are important when AOE is used in complex and dynamic cases in both closed and open agent environments. The ontology support, on the other hand, allows AOE to offer support for what we refer as semantic observation (see Section 2.2).

A soft-body is basically a collection of RDF statements associated with an entity. Since the soft-body is not managed by the entity itself, it is possible, for example, for an agent to be in a suspended state and still be observable. A newly created soft-body is always empty. At run-time, the content can be modified by both the entity itself and services residing on the MAS middleware layer. The modifications are executed using either agent or environment API provided by the AOE. The main difference between the two APIs is the scope of the modifications. Agent API only allows the entity to modify its own soft-body, whereas the environment API provides access to any soft-body in the system. This distinction is required for implementation of coordination mechanisms that are uncoupled from the behavior of the entities being coordinated. The parts of the observable state that are managed through different APIs are kept separate in order to avoid conflicts between them. So, while it is possible for an entity to observe its own soft-body as a whole using agent API, it cannot modify the state managed through the environment API.

As an example, let us considered an application where agent's trustworthiness in a competitive environment is determined by the value of its *reliability* property modeled as part of the agent's soft-body. The higher the value the more reliable agent is considered to be amongst its peers. Let us further assume that every available agent action has either positive or negative effect on the reliability of the agent. Now, one of requirements of the application is to manage the reliability of agents, based on actions they perform. If the *reliability* property was controlled by an agent itself, nothing would prevent it from making itself seem more reliable than it really is. Implementing *reliability* as an externally controlled property solves the problem. It externalizes the application logic related to the reliability management to an impartial component that is able to observe actions of agents and change the value of *reliability* property accordingly.

It is even possible for an entity to have observable, externally controlled properties that the entity is not even aware of. The soft-body can also contain both the state- and behavior related information. This means that the observable description of an action can be separated from the possible changes to the environment caused by that action. For example, the observer can first perceive that agent X is opening door Y, and later that the state of door Y changed from closed to open.

2.2 Observation Process

This section focuses on the act of observing and how it is addressed in AOE. The basic pattern for observation is same for all entities in MAS, but again, there are some differences in the scope of actions available through agent and environment APIs. The discussion in this section revolves around the concepts from the observation ontology [19]. According to Viroli et al [19], the general observation pattern is represented using four classes of systems: observers, sources, coordinators and internal observers. In the context of AOE, the source of observations is always the observable environment, but other roles can be played by any MAS entity.

AOE APIs allow entities to coordinate their observations by configuring the OE for *static* or *dynamic* observation. In MAS literature, this configuration process is referred as setting the foci [24] or creating a view [17]. The result of a static observation is a snapshot of the observable environment represented as observable items [19]. Dynamic

observation, on the other hand, allows entities to observe the evolution of the state and to receive observable events concerning either addition or deletion of data related their current interests [19]. In AOE, the observable items and events are always something that can be represented as one or more RDF statements. In case of a static observation, the observation configuration is removed from the AOE as soon as the observable items matching the query have been made available to the observer. For dynamic observation, the configuration stays active and keeps matching the events occurring in the OE until the configuration is explicitly removed by the observer. The internal observer, as defined in [19], can be modeled as a callback associated with the observation and thus the triggers are limited to the changes of the state of the OE.

In addition to static and dynamic observation, the environment API provides the capability to "push" observable items or events to selected entities, without any requirement for previously expressed interest or even consent from the receiver of the observation. This functionality is similar to the environment-controlled soft-body properties discussed in the previous section, and can become useful when implementing environment-based coordination infrastructures. It should be noted though, that the entities might not be under any obligation to react to the received observations. So the autonomy of agents is not compromised by such a push capability. We acknowledge the risk of a malfunctioning or a malicious component bringing down the whole system by spamming others with bogus observations. This kind of risk can be controlled using meticulous access control policies regarding API access.

Due to the semantic nature of AOE, MAS designers can take advantage of ontologies when designing observers and the OE. AOE infrastructure can use the inferred data when matching the state or events to the observation configurations. For example, by defining a class inheritance hierarchy for alarms, an agent is able to monitor all types of alarms in the system simply by configuring AOE to forward the events matching the root class of the hierarchy.

AOE allows an entity to dynamically focus its observational capabilities according to the task at hand. This does not however mean that the entity could always observe all the soft-bodies in system. One of the key features of situated MAS is the locality of interaction, which limits the interaction space of an entity to its observable neighborhood. In AOE, the locality can be modeled using observation rules.

2.3 Observation Rules

Rules in AOE are only used to limit the visibility of the information stored in the observable environment. In other words, rules control the read rights of properties and types stored in the soft-body. Observation rules cannot be used to modify or restrict the information stored as part of the soft-body.

The evaluation context for the rules is always the content of the observable environment. For example, a rule can limit the visibility of the files made observable by agent X to the agents that can be *observed* to be part of the same organization as agent X. We acknowledge the security issues related to this arrangement. If the access to some confidential information is associated with an agent-controlled value in the soft-body, it is easy for the observing agent to change that value to something that grants the access (to ”disguise”). One way to resolve the problem is to rely in access-granting rules on environment-controlled properties of agents, rather than on agent-controlled ones.

Every rule has a level and a priority. So called ”laws of nature”, or environmental rules, can be added using the environment API. These first-level rules always take the precedence over any other rules. First level rules can, for example, limit the visibility of confidential information based on properties controlled by the environment (see section 2.1). The second level rules are the ones added through the agent API. These can be, for example, task related rules that limit the visibility of some information made explicitly observable by the agent. Observation rules can be added dynamically at runtime, but there is currently no sophisticated way, beyond the rule priorities, to handle conflicting rules at the same level. The observation rules, in addition to the externally controlled soft-body properties and push observations (see section 2.2), are the third concrete tool for implementing the coordination mechanisms using AOE.

3 JADE Implementation

This section presents some details of the JADE [1] based implementation of the AOE framework described in the previous section. JADE middleware platform was selected as the basis for the implementation because of its relative popularity amongst the FIPA compliant platforms, in both the academic and industrial communities. Also, the current service-oriented internal architecture of JADE, which is based on the distributed version of the composition filters pattern [2], provides excellent facilities for extending the platform through custom kernel services. Since the role of the AOE is an infrastructural one, the most natural place to implement it is at the kernel level. This allows us to get a lower-level access to the data provided by the JADE framework by filtering the existing core services. Figure 2 shows the technical architecture of AOE in relation to the other services existing on the platform (Notification and Messaging services). The implementation is designed as a general plug-in and can therefore be integrated as part of any JADE application.

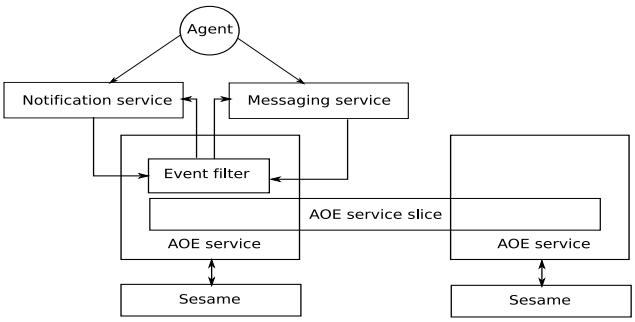

Figure 2. Technical architecture of AOE for JADE

The goal of the prototype implementation was to make both the state and the behavior of the agent observable. We decided to let the agent modify its observable state (soft-body) explicitly by using the agent API. For observable behavior, we wanted to make both a configurable list of JADE behaviors (practical actions) and the ACL messages (communication actions) always observable. These features were implemented using the environment API. The implementation uses a service filter to catch notification events created by agents in order to keep track of all active behaviors. Whenever a behavior matching the configuration of practical actions is added to the agent's scheduler, statement of the form *agentX aoe:does actionY* is automatically added to the agent's soft-body. We use a similar filter to capture messages from the messaging service, but instead of filling up the agent's soft-body by adding all the messages there, we use the push capability of the environment API to inform the possible eavesdroppers via observable events.

It would have been possible to implement similar functionality using just agents with the help of techniques utilized in Sniffer and Introspector agents shipped with JADE. However, the kernel level implementation removes the need for duplicating sent messages or wrapping event notification into ACL messages, because the observable data is directly available to the service.

We use Sesame [4], an open source RDF repository, for storing the state of the observable environment. The soft-body is implemented as two entity-related contexts, one for the agent and one for the environment API controlled data (see section 2.1). For example, when agent X adds to its soft-body the property *ex:currentTemp* with the value 30, the statement *X ex:currentTemp 30* is stored to the context *X/agent*.

4 Related Work

Since AOE is designed to only facilitate the observation process, it is not bound to, nor does it support any specific coordination framework or model. The idea is that the models and frameworks can be built or extended using AOE. Data-driven or subjective coordination [12], realized as actions selected by individual agents based on their perception of the state of the environment, can be implemented directly using the soft-body and observation rules. Examples of such coordination models are tag interaction [7] and property based coordination [26].

An objective coordination mechanism, which is uncoupled from the behavior of the agents [12], can be implemented using the environment API. As highlighted in the examples in the previous sections, environment API can be used to impose external control over the observations and observable state of an entity. This allows the creation of coordination frameworks similar to the tuple centers presented in [13] by Omicini et al.

5 Conclusions

The AOE framework presented in this paper provides a common medium for observation-based indirect interaction between MAS entities, such as agents and environmental services. The observability of an entity is based on its soft-body that acts as a container for entity-specific observable information. The soft-body is further divided into two distinct parts: one for the state managed by the owner of the soft-body and one for information managed by the environmental services.

The use of AOE provides a partial solution to the vertical and horizontal integration issues in MAS brought forward by Weyns in [21]. AOE allows the agents to coordinate their actions using a combination of middleware level services like digital pheromones [14], computational fields [11] or tags [7], all accessible through common observation medium, using RDF as the common data representation. AOE can also be used to bridge the gap between the modeling and the implementation of indirect interaction in MAS, because it eliminates the need to convert the interaction modeled as indirect into direct message passing between the interacting entities at the implementation phase [10].

We also reported on our implementation of AOE framework that is based on JADE agent middleware platform and Sesame RDF repository. We are now in the process of creating the first stable version of AOE for JADE. Our goal is to make this version available under an open-source license and continue its development as part of the JADE community effort. We also plan to evaluate the performance of the software framework and work on services and applications that take advantage of the AOE.

From the theoretical point of view, an interesting topic for further research is the possibility for observation-based learning with proper ontologies and the ability to ob-

serve/infer causal connections between agents' actions and changes to the environment caused by those actions.

Acknowledgments

This work is performed in UBIWARE project, which is supported by Tekes (Finnish National Agency for Technology and Innovation) and industrial partners Metso, Fingrid, Inno-W, Nokia, and ABB.

References

[1] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

[2] L. Bergmans and M. Aksit. Principles and design rationale of Composition Filters. In R. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect-Oriented Software Development*, pages 63–95. Addison-Wesley, 2004.

[3] R. Bordini, J. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. Wiley, 2007.

[4] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 54–68, London, UK, 2002. Springer-Verlag.

[5] R. Collier, G. O'Hare, T. Lowen, and C. Rooney. Beyond prototyping in the factory of the agents. In *Proc. 3rd Central and Eastern European Conference on Multi-Agent Systems (CEEMAS-03), LNCS vol. 2691*, pages 383–393. Springer, 2003.

[6] M. Dastani, B. van Riemsdijk, F. Dignum, and J.-J. Meyer. A programming language for cognitive agents: Goal directed 3APL. In *Proc. 1st International Workshop on Programming Multi-Agent Systems, LNCS vol. 3067*, pages 111–130. Springer, 2003.

[7] S. H. E. Platon, N. Sabouret. Environmental support for tag interactions. In *E4MAS*, pages 106–123, 2006.

[8] Foundation for Intelligent Physical Agents. *FIPA Abstract Architecture Specification*. Online: http://fipa.org/specs/fipa00001/SC00001L.pdf.

[9] A. Helsinger, M. Thome, and T. Wright. Cougaar: a scalable, distributed multi-agent architecture. In *Proc. IEEE International Conference on Systems, Man and Cybernetics. Volume 2*, pages 1910–1917, 2004.

[10] D. Keil and D. Goldin. Modeling indirect interaction in open computational systems. In *WETICE '03: Proceedings of the Twelfth International Workshop on Enabling Technologies*, page 371, Washington, DC, USA, 2003. IEEE Computer Society.

[11] M. Mamei, F. Zambonelli, and L. Leonardi. Cofields: a physically inspired approach to motion coordination. *Pervasive Computing, IEEE*, 3(2):52–61, April-June 2004.

[12] A. Omicini and S. Ossowski. Objective versus subjective coordination in the engineering of agent systems. In *Intelligent Information Agents - The AgentLink Perspective, LNAI vol.2586*, pages 179–202, 2003.

[13] A. Omicini and F. Zambonelli. Tucson: a coordination model for mobile information agents. In *In Proceedings of the 1st Workshop on Innovative Internet Information Systems*, pages 177–187, 1998.

[14] H. V. D. ParunaK, S. Brueckner, and J. Sauter. Digital pheromone mechanisms for coordination of unmanned vehicles. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 449–450, New York, NY, USA, 2002. ACM.

[15] E. Platon, N. Sabouret, and S. Honiden. Oversensing with a softbody in the environment – another dimension of observation. In *Proc. Work. Modelling Others from Observation at International Joint Conference on Artificial Intelligence*, 2005.

[16] A. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proc. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, LNCS vol.1038*, pages 42–55. Springer, 1996.

[17] K. Schelfthout, T. Holvoet, and Y. Berbers. Views: Customizable abstractions for context-aware applications in MANETs. In *Proc. Work. Software engineering for large-scale multi-agent systems*, pages 1–8, 2005.

[18] V. Tamma, C. Aart, T. Moyaux, S. Paurobally, B. Lithgow-Smith, and M. Wooldridge. An ontological framework for dynamic coordination. In *Proc. 4th Semantic Web Conference, LNCS vol. 3729*, pages 638–652. Springer, 2005.

[19] M. Viroli. On observation as a coordination paradigm: An ontology and a formal framework. In *Proc. ACM Symposium on Applied Computing*, pages 166–175, 2001.

[20] 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.

[21] D. Weyns, A. Helleboogh, T. Holvoet, and M. Schumacher. The agent environment in multiagent system: a middleware perspective. *International Journal on Multiagent and Grid Systems, Special Issue on Engineering Environments for Multiagent Systems*, 2008.

[22] D. Weyns and T. Holvoet. On the role of environments in multiagent systems. *Informatica*, 29(4):409–422, 2005.

[23] D. Weyns, H. V. D. Parunak, F. Michel, T. Holvoet, and J. Ferber. Environments for multiagent systems: State-of-the-art and research challenges. In *Proc. 1st Work. Environments for Multi-Agent Systems, LNAI 3374*, pages 1–47, 2004.

[24] D. Weyns, E. Steegmans, and T. Holvoet. Towards active perception in situated multi-agent systems. *Applied Artificial Intelligence*, 18(9–10):867–883, 2004.

[25] M. Wooldridge. Agent-based software engineering. *IEE Proceedings of Software Engineering*, 144(1):26–37, 1997.

[26] M. Zargayouna, J. S. Trassy, and F. Balbo. Property based coordination. In I. J. Euzenat and J. Domingue, editors, *Artificial Intelligence: Methodology, Systems, Applications*, volume 4183 of *Lecture Notes in Artificial Intelligence*, pages 3–12. Springer Verlag, 2006.