Electronic Communications of the EASST
Volume 37 (2011)

Workshops der wissenschaftlichen Konferenz Kommunikation in Verteilten Systemen 2011 (WowKiVS 2011)

Paradigm-Independent Engineering of Complex Self-Organizing Systems

Michael Zapf

12 pages

Guest Editors: Horst Hellbrück, Norbert Luttenberger, Volker Turau
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Paradigm-Independent Engineering of Complex Self-Organizing Systems

Michael Zapf

zapf@vs.uni-kassel.de,
Distributed Systems Group
Universität Kassel, Wilhelmshöher Allee 73,
34121 Kassel, Germany

Abstract: We present EPAC, a novel approach to design and engineer systems of autonomous components. EPAC is based on the Model-Driven Architecture concept and subsumes concepts from different design paradigms that are related to the construction of complex systems, like multi-agent systems, multi-robot systems, or sensor networks. The design process introduces a generalized metamodel and starts at a high abstraction level which defines model elements for autonomous components, for structures, relationships, and behavior. We describe the stages of the modeling process, pointing out the specific differences to existing approaches.

Keywords: Model-driven software engineering, MDA, autonomous components, methodology

1 Introduction

During the last decade, agent system and agent application engineering were gradually replaced by the research on agent system design methodologies. The idea behind such a methodology is to propose a practical, efficient, and application-independent recipe to design agent software. We can also find many fields of research with noticeable parallels to agent systems: For instance, sensor networks are running a distributed application program which makes them collect values, exchange them with neighboring nodes, or reorganize the network to maintain connectivity [9].

Autonomous cooperative robots have also been a hot topic in research during the last decade. Many projects are involved in creating teams of robots which shall solve a task cooperatively, the most popular one being RoboCup where teams of robots play soccer; in other projects they are part of rescue scenarios.

All of the aforementioned research areas have developed own tools and ideas to deliver adequate solutions. These ideas imply specific properties of their target environment, mainly driven by capabilities and constraints of the objects and devices. The way how problems are conceptualized, the models, and practices that are employed are comprised in a general notion what we call a *paradigm*.

Systems situated in different paradigms have specific properties which require an appropriate modeling. On the other hand, methodologies for these various areas have become strictly separated; the application of an agent-related methodology for sensor networks does not seem reasonable in the first place. We found many similarities within scenarios that have always been

considered as being located in different paradigms, as soon as we get to higher layers of abstraction:

- The function of the complete system depends on the interoperation of a certain set of self-contained components, seen as subsystems of their own. This allows for the traditional divide-and-conquer strategy of solving complex problems.
- The subsystems show a high level of autonomy; activities originate from their own decisions, possibly following an internal plan, and need not necessarily be triggered from a central control.
- The overall system involves so many components or shows a high level of complexity that self-organization becomes a required feature of the system.

We argue that while the specific features can be specifically handled, there is a need for a common methodology on an abstract level which offers the potential to share approaches for different paradigms.

In this article we present a new approach for engineering which we call *EPAC* (Engineering Process for systems of Autonomous Components). It starts at a very abstract level, defining structures and behaviors. Next, the model designer specializes the models for a paradigm, then for a platform.

Section 2 provides background information by summarizing features of existing methodologies. In Section 3 we describe the general concept of our approach, and in Section 4 we explain how to perform the overall process. Section 5 summarizes the basic ideas and expected effects of applying our approach, and Section 6 provides conclusions and an outlook on open issues.

2 Related work

Within the last 15 years, numerous approaches for the design of complex systems have been proposed, of quite different nature. For instance, there are paradigms like object-oriented programming, component-oriented programming, aspect-oriented programming, or agent-oriented programming; or there are tools like UML which help to create models of the systems to be designed.

Methodologies are commonly understood to support a wide scope within the development process, with some of them focusing on the earlier phases of analysis and design, others supporting the detailed elaboration and creation of software, and also some of them attempting to guide the complete process from analysis to implementation. Methodologies are often grounded on a theoretical model of the complete process and of the entities and relations that are of interest in the target environment. Researchers in agent software engineering widely agree (as usually found in the introduction of the description of their methodologies) that the traditional tools and methodologies fail to represent agent properties in their theoretical model and so essentially make the designer create concepts which are inherently not agent-oriented.

We have a brief look at some well-known AOSE methodologies in order to be able to evaluate the specifics of our approach.

2.1 Tropos

Tropos [1, 6] is a methodology which targets many phases in the development process, spanning from early requirements analysis to detailed design. The Tropos concept of designing a complex system emphasizes the *actor* as a more generalized notion of a contributing party or component within the system which encompasses human or organizational stakeholders, parts of the environment, and certainly also components which eventually end up as agents to be implemented. Diagrams like the *actor diagrams* and the *goal diagrams* allow to intuitively conclude which activities are required to fulfill some goal.

Despite the general approach, Tropos is essentially a methodology to design agent-oriented systems in the first place. The central idea of Tropos lies within the concept of goals which need to be fulfilled by active contributors, the actors. Tropos assumes a target infrastructure which supports a goal-oriented processing of autonomous components; it does not comment on realizing the system in system environments that probably cannot afford such a high-level processing.

2.2 PASSI

PASSI [2, 3] centers the design on roles and tasks to be handled, creates a society model, and approaches the implementation by dividing functionality among those parts dedicated to single agents and those for agent groups. The core concept of PASSI is to understand the agent as a significant unit in both conceptual and implementation views [2]. An agent may play different roles throughout its lifetime, which determines the agent-internal behavior as well as the interactions within the group, so PASSI explicitly provides modeling support for both views. PASSI makes strong use of various UML diagrams and concepts like stereotyping.

Many scenarios discussed for PASSI assume a FIPA-based agent implementation, and the proposed metamodel [3] includes the FIPA agent within its problem domain. Nevertheless, the methodology is also applicable for non-FIPA and also non-Java environments [2], but it still targets at the creation of applications with a clear agent-oriented notion.

2.3 Gaia

The Gaia methodology was proposed by Franco Zambonelli, Nicholas Jennings, and Michael Wooldridge in 2000, with some enhancements in 2003 [12]. It assumes a multi-agent system to consist of a set of agents playing certain roles in an environment which is determined by organizational rules. Each role implies a set of responsibilities. By repeated refinements, Gaia produces a set of agent types and service types which represent the agent functionalities.

Gaia introduces only few formal elements, trying to avoid a too technical point of view, like *role diagrams* and *protocol diagrams*. It is a comprehensive way of approaching the design of a complex system using clearly defined phases, however, staying on a comparably abstract level, and only providing high-level guidelines for a later implementation phase. Gaia at its core does not deliver an adequate support for model-driven software engineering, particularly with respect to model and code transformation.

3 An Engineering Process for Systems of Autonomous Components

The prevalent strategy for encouraging system designers to make use of the AOSE methodologies is to advocate an "agent view" on the system, understanding it as a collective of cooperative, autonomous agents. In our approach, we suggest another, dual view, by abstracting from specifics of technologies as far as possible, while still retaining the capability to create models from formal metamodels. Ideally, these *paradigm-independent* models (PAIM) can be re-used in parts or in whole for scenarios within quite different target environments. The PAIM forms another abstraction layer prepended to the modeling layers of the well-known MDA [8] modeling concept.

3.1 The PAIM metamodel

The core of our approach is the notion of a generalized concept of an autonomous computing entity which we call *ACE* for short. An ACE is an entity that represents a component of the system with the following properties:

- The ACE is perceived as a subsystem with a clearly defined system boundary. For each ACE it makes sense to distinguish the ACE from its environment as the rest of the system where it is embedded.
- The ACE behavior, as perceived during its life cycle, is considered as *its own* behavior. ACEs are capable of initiating activities.
- ACEs can interact with each other or with other parts of the embedding system.
- An ACE may consist of identifyable autonomous parts (sub-ACEs) which, by their cooperation, constitute the structure and behavior of the ACE.

These properties may sound familiar to agent researchers. There are three main reasons why we suggest to use the term ACE at this level, not agent: First, we explicitly target *computational artifacts*, while some other metamodels suggest to include humans in the agent or actor semantics. Second, the term better reflects the *independence* from the agent paradigm at this level, allowing to apply this metamodel not only to software agents but also to related areas. Third, an agent-oriented model eventually suggests the designer to map the model concept to a single agent entity. However, it is not always clear what parts of the system are reasonably designed as having agent properties. There may also be situations where we have collections of independent entities which show a single group behavior. For this situation we define the semantics of an ACE model element as representing a possibly substructured, self-contained, autonomous entity of the system, following a holonic view. Figure 1 gives an example how the ACE model element may be translated to different paradigms and platforms, and how general concepts map to different, yet comparable terminology.

The *Model-Driven Architecture*, or short MDA [8], has been introduced by the Object Management Group in 2001 as a special realization of the MDD notion, defining four levels of abstraction: computational-independent models (CIM), platform-independent models (PIM), platform-specific models (PSM), and the platform-specific implementation (PSI). The EPAC process that

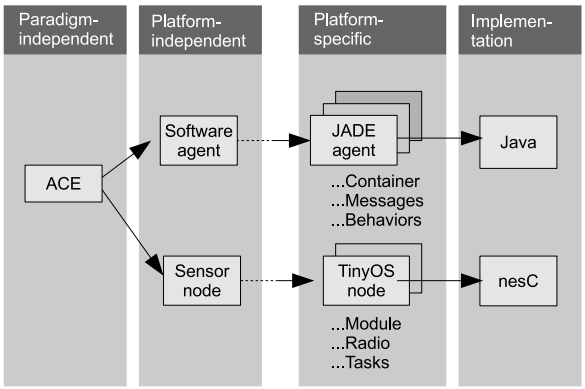

Figure 1: Specializing towards paradigms and platforms

we are about to describe is not intended to introduce another theoretical model. Instead, its primary objectives are three-fold:

- encourage the re-use of structural and behavioral patterns on an abstract level, separating them from paradigm-related assumptions;
- explicitly integrate the concept of self-organization as a model element for group behavior which has been difficult to combine with existing techniques so far, and
- support the designer by the inclusion of model-driven software development techniques for creating complex systems of autonomous components.

As our approach is ignorant about metamodels on the paradigm level, we expect that other methodologies may be cooperatively used. Concretely, during the modeling process we may have collected a set of features and responsibilities associated to an agent role; these findings may serve as input to a special AOSE modeling technique like Gaia. A complementary usage would be to switch between the methodologies multiple times as required, while a successive usage means to start with one methodology and continue with another one to the end or vice versa.

3.2 Features and Responsibilities

The ACE behavior on a abstract level is modeled as a collection of *features* that the ACE may expose during its lifetime. Related features are then grouped into responsibilities, a concept already known from other methodologies like Gaia. We distinguish between two classes of features:

- Primary features (application-related) which represent the proper goals of the system and its design.
- Autonomic features, covering all kinds of behavior that are related to self-organization and similar self-properties.

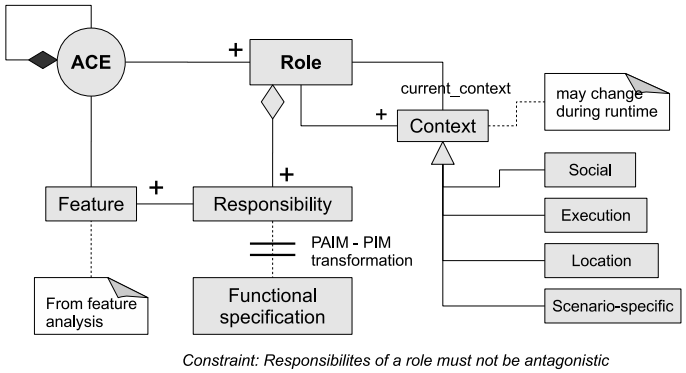

Figure 2: PAIM metamodel

The term *autonomic features* refers to properties which are ensured by the system during runtime as a background activity. In contrast, primary features are located in the foreground and cover the set of actual requirements for the system, immediately related to the scenario.

The background activity is specifically focused on structure-preserving actions, configurations, adaptations, corrections and similar; these areas of activity are also main topics of the *Autonomic Computing* concept initiated by IBM [9]. As autonomic features are intended to keep human operators out of the control loop, they infer self-properties in the involved components. Realizing self-properties for systems requires the creation of closed control loops, for which we know already prepared patterns like MAPE(K) or OC cycles [10]. Control loops may also be hierarchically organized.

Within the design of such systems, particular attention must be paid to the fact that various responsibilities may have antagonistic nature, which means that they cannot be handled by the same entity at the same time. An example could be a multi-agent system with mobility and load balancing: Decisions from one part likely interfere with the other part. Situations like these give reason to introduce more than one role, and so to avoid to group together incompatible responsibilities.

The EPAC process envisages that ACEs may be nested and thus form holarchies. One one hand, this allows the designer to compose ACEs by more elementary ACEs which follow their own, possibly contradicting plans. The higher-order ACE will be required to employ some policy to coordinate the subcomponents. The highest-order ACE, in this model, would be the overall system, encompassing all components involved in the scenario. On the other hand, we can treat higher-level ACEs – containing other ACEs as parts – as bearers of responsibilities.

3.3 Roles and contexts

In the metamodel, a role is a consistent union of responsibilities, where no two responsibilities have an antagonism relationship. ACEs may take several roles during their lifetime.

A context is commonly understood as the part of a global system in which the actual system of interest is situated, and which influences its behavior. Contexts may consist of other complex and intelligent entities, of environmental aspects like spatial location, of physical parameters which

the system is exposed to, or logical conditions like system configurations. A role is associated with one or more contexts, reflecting the possibility that the execution of a role may trigger context changes.

Context changes may happen if the environment changes, either by inherent change (reconfiguration), or by a relocation of the system into another environment. These context changes are usually triggers for behavioral adaptations. Context changes, in this most abstract notion, do not imply specifics of a paradigm, and thus are consequently situated in the PAIM level. One example of a PAIM/PIM mapping could then be to map context changes to agent migration in the mobile agent paradigm, declaring migrations as deliberately triggered context changes. This, in turn, allows to *lift* the migration concept up one level. The restructuring of ACEs, probably as a result of self-organization, is modeled by contextual changes for sub-parts. In that view, various *configurations* may be modeled, and blackbox elements may be employed to abstract from deeper detail, for instance, if the internal structure will be created by self-organization.

With the contexts situated at the PAIM level, we can create behavioral models (e.g. using state machines or activity diagrams) which include the dynamics of environmental changes or internal changes at a higher level. One consequence of such a context change could be the replacement of behavioral parts as a result of an adaptation activity [5].

4 The EPAC process

From the metamodel we can derive a sequence of activities which the system engineer has to follow in order to successfully create a system, according to our approach.

4.1 Overview

As Figure 1 suggests, the EPAC process starts at the highest abstraction level: the paradigm-independent level, that is, analysing the overall system as a collection of interacting ACEs. As ACEs have a holonic nature, it is reasonable to view the complete system as *a single ACE* and then refine its substructure.

Following the metamodel, features need to be collected which are derived from the actual scenario. The features have to be sorted into two sets: Primary features are those that represent the foreground activity, while autonomic features are all features that ensure some desired self-property of the system.

Figure 3 shows that these features are then grouped into responsibilites. From this stage, the separate handling is over, and the responsibilities are grouped into roles.

Roles can then be assigned to one or more ACEs. At this point we have to decide whether we can associate the roles to the currently considered ACE, or whether we need to break apart the system into smaller subsystems, due to a conflict between the roles. For instance, a conflict will arise if the execution of the roles requires to be at several locations at the same time. In that situation, a distribution of the roles among several sub-ACEs is required. The overall system will then consist of the union of those sub-ACEs.

Note that the process may now recurse if required. Especially in the case where the ultimate realization of an ACE may spread over separate network locations, a finer granularity is required.

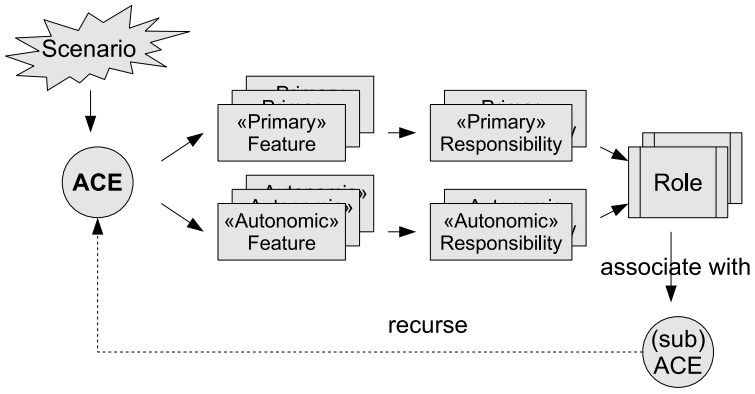

Figure 3: Overall process at the PAIM level

This reflects the traditional notion of divide-and-conquer. Then again, we can isolate primary and autonomic features, and so on.

It is important to note that while ACEs may be split into sub-parts, this does not imply that all features need to be distributed among them. For example, the designer may consider to associate the autonomic feature of self-organization to the complete ACE. In this case it would not make sense to break down this feature to the subparts but instead to employ appropriate techniques like evolutionary algorithms, as shown for example in [11, 16, 15], in order to create a group behavior. The most important responsibility of the designer is to verify that the generated behavior does not interfere with the dedicated roles of the sub-ACEs.

4.2 Application of the PIM-PSM-PSI chain

Up to here, the design process ignored any paradigm or platform-specific properties. At some point we will not be able to elaborate further on the system without inferring features from the underlying paradigm and platform. For instance, we need to decide whether ACEs can relocate themselves in the network, thus being mobile agents. Still, we need not select a target platform, like JADE or AMETAS.

The fact that a PAIM does not specialize on one paradigm does *not* mean we can freely choose any possible paradigm for the same scenario, as every paradigm has its specific application areas. An application inherently suited for mobile agents will possibly not run in a sensor network. The idea of the EPAC process is to allow to unify the high-level modeling process, from where the modeling can continue by the appropriate paradigm.

If the design has become sufficiently detailed, making use of structural and behavioral models (and possibly other methodologies), the designer may now perform the first transformation step: **Transform the paradigm-independent models to a target paradigm** like, for instance, *software agents*. The transformation to the target platform causes certain terminology to be imported into the model. For instance, we can map ACEs to software agents, interactions to asynchronous messages, and locational contexts to places.

At this point we still avoid to map the terminology to more concrete terms which we know from specific agent systems. The designer now has a detailed description of how the scenario

could be designed under the software agent paradigm. From here, various specialized methodologies may take over, or we directly continue with the specialization. It is also possible to include another methodology in this process, delivering a suitably refined platform-independent model, and then return here.

According to the MDA concept, the next step would then be the PSM transformation: **Transform the platform-independent model to a target programming / application platform**. Finally, the PSM may be populated with further specific diagrams (like state machines) which contribute to the final code transformation. As we rely on UML models, the transformation can be automatized at several points in the process. The automatic conversion amounts to write suitable transformation scripts or rulesets, using, for instance, XSL transformation or *MOF Model To Text Transformation* [7].

It should be noted, however, that this process will not run from start to end without any contribution by the human designer. At each stage of the transformation process, the designer will have to add information which is specific for the respective paradigm or platform, as the more abstract models naturally lack this information. The amount of information within the model is increased by each transformation step, and not every structural and behavioral feature may be adequately represented within abstract models. Instead, the process is intended to provide automated transformations whereever possible, and hence to guide the specification of models and the implementation by providing common mappings, delivering *skeleton models* and *code*.

4.3 Implementation

At the end of the design process, we find a number of diagrams as a result, representing platform-specific models. These models can represent structural features or behavioral features, also containing commonly used diagram types like state machines diagrams.

Figure 4 shows a part of the XSL transformation used to convert a state machine diagram into JADE code. Obviously, this transformation will only produce a code skeleton which still needs to be completed. One straightforward extension to this sample is to include transition triggers and effects as message exchange actions. Using another XSLT document we were able to immediately create code for a second agent system, AMETAS [14, 13]. XSLT is not the only possible choice for a transformation language; other examples are *MOFScript*[4] or *MOF Model To Text Transformation* [7] which are specifically targeted at creating code from XMI documents and provide a better readable transformation code than XSL.

5 Rationale of the process

Viewing the basic notion of our approach just as an iterated model specification does not cover the overall concept appropriately. We summarize the core ideas of our approach at this point.

5.1 Formalized metamodel

Models in EPAC are formulated using the UML language. This allows the designer to make use of existing tools, like Eclipse UML tools, Papyrus UML, or Enterprise Architect. As UML is a formal metamodel there are practical ways of automatic verification of user models. UML

```
<![CDATA[    public ]]><xsl:value-of select="$agname"/><![CDATA[() {
        // Put your initialization code here
    }
    protected void setup() {
        FSMBehaviour ab = new FSMBehaviour();
        ab.registerFirstState(new ]]>
            <xsl:value-of select="subvertex[@xmi:type='uml:Pseudostate']
            /@name"/>
            <![CDATA[(this), "]]>
            <xsl:value-of select="subvertex[@xmi:type='uml:Pseudostate']
            /@xmi:id"/><![CDATA[");]]>
        <xsl:for-each select="subvertex[@xmi:type='uml:State']"><![CDATA[
            ab.registerState(new ]]><xsl:value-of select="@name"/>
                <![CDATA[(this), "]]><xsl:value-of select="@xmi:id"/>
            <![CDATA[");]]>
        </xsl:for-each>
        <xsl:for-each select="transition"><![CDATA[
            ab.registerTransition("]]><xsl:value-of select="@source"/>
            <![CDATA[", "]]>
            <xsl:value-of select="@target"/><![CDATA[", TRANS]]>
            <xsl:value-of select="fn:position()"/>
            <![CDATA[);]]>
        </xsl:for-each><![CDATA[
        addBehaviour(ab);
    }
...
```

Figure 4: Excerpt from the XSL transformation UML/JADE

may be used to formalize a metamodel to be used for our PIM models, or we can use *Profiling* to introduce variants of existing model elements. By using the XMI diagram export features, we can transform models to XML documents in a standardized way and then algorithmically transform them into new XML documents, for instance, by using XSL transformations, as shown above.

5.2 Abstract modeling

The EPAC methodology encourages to abstract further from the specific terminology up to the point where the designer considers autonomous computing entities to interact with each other, and to form a collective which provides a certain functionality. From here on, with sufficiently detailed models, specifications may be derived by the transformations. In the process of modeling, features which belong to self-organization and related concepts are separated from foreground (primary) features which are the design objectives of the system. By this separation, techniques like triggering emergent features may become a part of the engineering process.

5.3 Methodology reuse

The EPAC modeling concept may be understood as *orthogonal* to existing methodologies. It is compatible with concepts of organizations, roles, responsibilities, acquaintances and similar notions introduced by other methodologies like Tropos, Gaia, or PASSI. By its abstract view, it may encourage designers of related domains to apply methodologies which were originally targeted towards agent systems. One example could be to design a self-organizing sensor network which intelligently reorganizes its structure, depending on reachability, making use of behavior patterns which proved to be efficient for agent societies.

6 Conclusions

The EPAC process described in this article is intended to broaden the scope of existing methodologies in the area of complex system design in order to allow a designer to utilize suitable concepts, patterns, and processes despite the fact that they were originally intended for another area. The core concept is the setup of a common metamodel which is useful to describe features in the related areas, and transformations which translate models from the paradigm-independent via the platform-independent to a platform-specific representation, which can then eventually be transformed to code.

Agent-oriented software engineering methodologies are, in most cases, already generic enough to be employed in related research areas as well, merely requiring the designer to adopt an ”agent view” on the system to be created. However, we find only little evidence that AOSE has found a broader audience outside of the agent community. This lack of acceptance may be remedied in our view by avoiding to fix the modeling to one paradigm at an early stage. Rather, following the EPAC concept, we suggest to start modeling at a very abstract level, avoiding to assume paradigm-specific properties, which should encourage designers to approach their design in an unbiased way, without committing to a paradigm. That way, we expect that common patterns for structures and behavior can be recognized more easily and predefined modeling concepts can be applied. Later, the models can be transformed to a paradigm, and then to specific platforms, following the MDA concept.

Various parts of EPAC are still work in progress, and we expect amendments in all parts of the overall process as we complete the metamodel definition, the transformations to expected target platforms, and evaluate the applicability to further paradigms or platforms. We are currently working on the transformation of model diagrams between the process stages and for the ultimate code generation. One especially interesting point is the combination of different diagrams in the transformation process to form a single result model or code.

Bibliography

[1] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. TROPOS: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, May 2004.

[2] M. Cossentino and C. Potts. PASSI: a process for specifying and implementing multi-agent systems using UML, 2002. http://www.cc.gatech.edu/classes/AY2002/cs6300_fall/ICSE.pdf.

[3] Massimo Cossentino, Salvatore Gaglio, Luca Sabatucci, and Valeria Seidita. The passi and agile passi mas meta-models compared with a unifying proposal. In *CEEMAS*, volume 3690 of *Lecture Notes in Computer Science*, pages 183–192. Springer, 2005.

[4] Eclipse Foundation. MOFScript Home Page, 2010. http://www.eclipse.org/ gmt/mofscript/.

[5] Kurt Geihs, Roland Reichle, Michael Wagner, and Mohammad Ullah Khan. *Modeling of Context-Aware Self-Adaptive Applications in Ubiquitous and Service-Oriented Environ-*

ments, pages 146–163. Software Engineering for Self-Adaptive Systems. Springer-Verlag, Berlin, Heidelberg, 2009.

[6] P. Giorgini, M. Kolp, J. Mylopoulos, and M. Pistore. *The Tropos Methodology: An Overview*. Kluwer Academic Publishing, 2004.

[7] Object Management Group. MOF Model To Text Transformation Language Home Page, 2008. http://www.omg.org/spec/ MOFM2T/1.0/.

[8] Object Management Group. OMG Model Driven Architecture Home Page, 2009. http://www.omg.org/mda/.

[9] J. Kephart and D. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1), Jan 2003.

[10] Urban Richter, Moez Mnif, Jürgen Branke, Christian Müller-Schloer, and Hartmut Schmeck. Towards a Generic Observer/Controller Architecture for Organic Computing. In *INFORMATIK 2006 - Informatik für Menschen!*, volume P-93 of *LNI*, pages 112–119. Bonner Köllen Verlag, October 2006.

[11] Thomas Weise, Michael Zapf, Mohammad Ullah Khan, and Kurt Geihs. Genetic programming meets model-driven development. In *7th International Conference on Hybrid Intelligent Systems (HIS 2007)*. IEEE Computer Society, 2007. http://www.it-weise.de/documents/files/WZKG2007DGPFg.pdf.

[12] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370, 2003.

[13] M. Zapf. Type-based mediation of mobile agents. In *International ICSC Congress: Intelligent Systems & Applications ISA '2000, Wollongong, Australien*, volume 1, pages 236–241. ICSC Academic Press, December 2001.

[14] Michael Zapf. *Typisierung autonomer Softwareagenten*. PhD thesis, Johann Wolfgang Goethe-Universität, Frankfurt/Main, Germany, April 2002. German only; online available: http://www.mizapf.de/work/mzapfdiss.pdf.

[15] Michael Zapf and Thomas Weise. Applicability of emergence engineering to distributed systems scenarios. (TechRep 2008, 5). Presented at EUMAS '08; http://www.vs.uni-kassel.de/publications/2009/ZW09.

[16] Michael Zapf and Thomas Weise. Can solutions emerge? In *The third International Workshop on Self-Organizing Systems (IWSOS '08)*, volume 5343 of *Lecture Notes in Computer Science (LNCS), LNCS Sublibrary: SL 5 – Computer Communication Networks and Telecommunications*, pages 299–304. Springer, dec 2008.