IEEE distributed systems Expert-authored articles and resources ONLINE
November 2008 (vol. 9, no. 11), art. no. 0811-mds20080y0002

Published by the IEEE Computer Society

Model-Driven Development

Model-driven Development and Adaptation of Autonomous Control Applications

Helge Parzyjegla, Michael A. Jaeger, and Gero Mühl • Berlin University of Technology
Torben Weis • University Duisburg

In the near future, we expect a variety of actuator and sensor nodes to populate the environment.[1] Our vision is driven by rapid progress in hardware development, such as miniaturizing computing devices. On the software side, however, application development for such a setting is challenging for several reasons. One is the heterogeneity of devices and networking technologies, which vastly increases application development complexity. Other reasons include frequent reconfigurations and communication faults (such as network partitioning) that have to be handled appropriately. These reasons present problems for software developers—they can't predetermine configurations or anticipate all of the potential runtime faults that might occur. Furthermore, users might not be willing or able to handle complex configuration or communication issues—they want to install their devices and use applications. Thus, devices and applications must be able to work as autonomously as possible, with little to no manual user intervention. Specifically, applications must be able to adapt at runtime to a changing environment and recover from faults.

Here, we elaborate on ideas we presented in a previous paper[2] and discuss in more detail a model-driven approach to developing and adapting autonomous control applications. In contrast to conventional approaches, we use the application model not only for design and deployment but also for dynamically adapting the application at runtime. This is in line with research that emphasizes the importance of exploiting the latent knowledge contained in models at runtime.[3] Our goal is to empower application developers to create self-organizing and robust actuator and sensor network (AS-Net) applications with minimal expert knowledge. Our work is part of the Model-Driven Development of Self-Organizing Control Applications project (www.kbs.tu-berlin.de/modoc).

Related Work

The work in this paper is part of the Organic Computing Initiative (OCI), which aims to develop adaptive and self-organizing applications. Several OCI projects use the observer/controller design pattern.[1] It's closely related to the Monitor, Analyze, Plan, and Execute cycle proposed by IBM's Autonomous Computing Initiative[2] and resembles several aspects of control loops that seek to keep a system's output near a given reference set point. In the observer/controller pattern, observers monitor the system and take corrective actions when necessary. However, relying solely on runtime information isn't always sufficient because the application itself might have individual requirements that its model specifies. By exploiting information gained from monitoring the environment at runtime as well as from the model, we fill the gap between other approaches that focus only on design time[3] or runtime.[4]

The research carried out in the reflective middleware domain is close to our approach. Reflective middleware possesses a representation of its behavior and structure in which the representation is causally connected to the actual system—that is, changes in the representation have a direct impact on the behavior and structure of the system and vice versa.[5] With our approach, the task of adapting the system is transparent for application developers and handled by our algorithm stack. We tackle the challenge of realizing reconfiguration at runtime to provide seamless adaptation without

interrupting the system service separately. Regarding the network topology's reconfiguration, we can build on earlier work.[6,7]

Architectural description languages (ADLs) are used to describe how software systems are organized with respect to their inner structures (components, interfaces, communication, and so on).[8] They're mainly used for communication, validation, and analysis, as well as prototype generation. It's also possible to specify self-managing architectures in terms of dynamics.[9] However, our goal is to hide these details from developers to empower them to create autonomous applications with little to no expert knowledge in the self-management domain. Therefore, on the one hand, we restrict developers to applications that are based on the concept of roles; on the other hand, we provide an integrated toolchain that generates distributed implementations for applications that exhibit recovery guarantees in case of transient faults. We believe that the role-based approach is easier to follow than a general component-based approach, which provides less abstraction to developers. This way, our approach is similar to that of a service-oriented architecture (SOA), in which systems are built from services orchestrated to provide a required functionality. However, a role only exists in an application context, while services might be reusable and stand on their own.

Jini (www.jini.org) and UPnP (www.upnp.org) already provide infrastructures for home and corporate network environments. They focus mainly on discovery and usage of applications or devices, whereas we concentrate on the development of distributed applications. Furthermore, it's not clear yet if these infrastructures are themselves self-stabilizing such that they meet our requirements. If shown to be self-stabilizing, it might be possible to use them for the lower communication layer and the cooperation between roles on different devices.

References

1. U. Richter et al., "Towards a Generic Observer/Controller Architecture for Organic Computing," C. Hochberger and R. Liskowsky, eds., Lecture Notes in Informatics, vol. P-93, Köllen Verlag, Sept. 2006, pp. 112–119.
2. J.O. Kephart and D.M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, 2003, pp. 41–50.
3. D. Skogan, R. Grønmo, and I. Solheim, "Web Service Composition in UML," *Proc. 8th IEEE Int'l Enterprise Distributed Object Computing Conf.* (EDOC 04), IEEE Press, 2004, pp. 47–57.
4. F. Kon et al., "The Case for Reflective Middleware," *Comm. ACM*, vol. 45, no. 6, 2002, pp. 33–38.
5. G. Coulson, "What is Reflective Middleware?" *IEEE Distributed Systems Online*, Dec. 2001, vol. 2, no. 8.
6. M.A. Jaeger et al., "Reconfiguring Self-Stabilizing Publish/Subscribe Systems," *Proc. 17th IFIP/IEEE Int'l Workshop on Distributed Systems: Operations and Management* (DSOM 06), Springer, 2006, pp. 233–238.
7. H. Parzyjegla, G. Mühl, and M.A. Jaeger, "Reconfiguring Publish/Subscribe Overlay Topologies," A. Hinze and J. Pereira, eds., *Proc. 5th Int'l Workshop on Distributed Event-Based Systems* (DEBS 06), IEEE Press, 2006, p. 29.
8. P.C. Clements, "A Survey of Architecture Description Languages," *Proc. 8th Int'l Workshop on Software Specification and Design* (IWSSD 96), IEEE Press, 1996, p. 16.
9. J.S. Bradbury et al., "A Survey of Self-Management in Dynamic Software Architecture Specifications," *Proc. 1st ACM SIGSOFT Workshop on Self-Managed Systems* (WOSS 04), ACM Press, 2004, pp. 28–33.

Application Model and Model Transformation

An application scenario that provides several opportunities for innovation but also many challenges for the development, deployment, and administration of applications is the e-Home. An e-Home aims to provide users with an intelligent environment that consists of plug-and-play applications, which require minimal manual intervention. An important prerequisite to achieving this goal is to equip application developers with a framework that enables them to focus on functionality instead of technical issues such as platform heterogeneity and networking technologies. This also encompasses

distributing the application to available devices and reconfiguring the network when devices are added or removed.

Within the scope of the Model-Driven Development of Self-Organizing Control Applications (MODOC) project (www.kbs.tu-berlin.de/modoc), we're developing a toolchain that comprises an easy-to-learn modeling language, a graphical development environment, and a model transformation process that encapsulates the necessary expert knowledge (regarding self-stabilization through leasing and soft state, for example) to deal with issues such as heterogeneity and self-organization at design time and runtime. Beyond this classical model-driven approach, application development is based on the concept of roles to ease the development of distributed applications and to add basic self-stabilizing and self-organizing properties to the whole system.

Figure 1 summarizes the development and transformation process. The starting point is the application model provided by the application developer. The application model that's created with the graphical modeling language is transformed in multiple steps to finally generate code for diverse target platforms. In the first transformation step, the model is analyzed and split into roles that cooperatively realize the application's functionality. The arrows in figure 1 show communication among individual roles. Each role is inspected to derive the requirements that a node must fulfill to execute this role. This could be, for example, a special set of sensors that a node must possess to detect a certain class of events. Furthermore, roles are equipped with self-stabilizing algorithms (for leader election or consensus finding, for example) that aid self-organization at runtime. These algorithms are taken from our algorithm toolbox, which contains different self-stabilizing algorithms for many purposes and provides them to the model transformers in a parameterizable fashion. Finally, code is generated for the target platforms while it's ensured that every node is at least equipped with self-stabilizing algorithms for inter-role communication and dynamic role assignment.

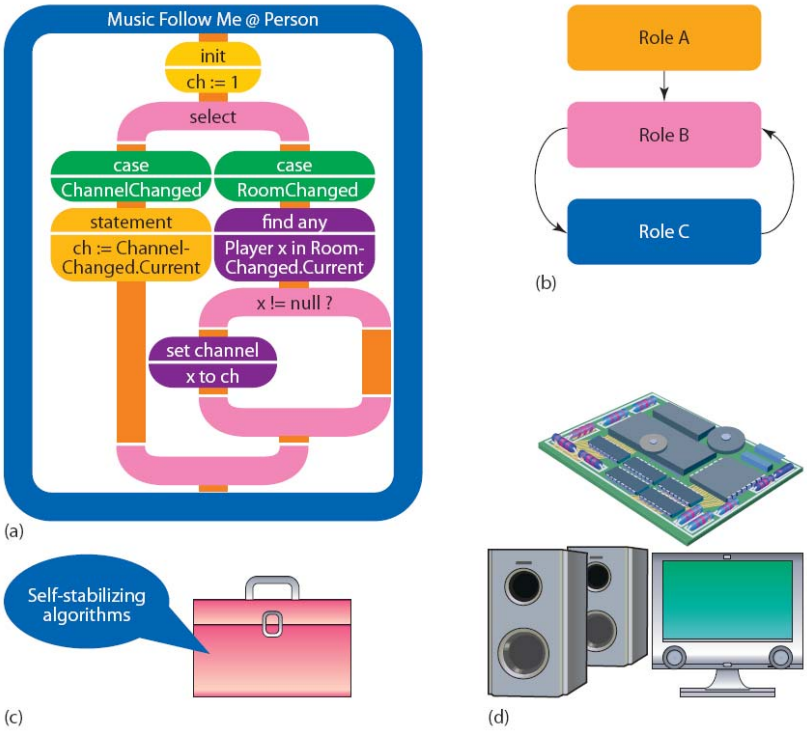

Figure 1. Multi-step transformation process. (a) Application model. (b) Intermediary role model. (c) Augmenting the roles with self-stabilizing algorithms from the algorithm toolbox. (d) Code generation for target platforms.

Modeling Language

We tailored our graphical modeling language to the development of pervasive applications while trying to keep complexity at a moderate level. Every programming construct has a visual representation enabling even novice programmers to familiarize themselves quickly and design their first applications.[4] The modeling language is built on concepts derived from the π-Calculus.[5] Thus, applications consist of several rather small and independent processes that asynchronously exchange messages using named ports. Usually, a process describes a particular workflow that's executed when certain events are detected. Note that a single process in the model might lead to several processes at runtime. Because the modeling language hides aspects of distribution, a workflow is allowed to query arbitrary sensors, relate to state information stored on different devices, and operate arbitrary actuators for manipulating the environment. Such a workflow requires specific program code to be executed on each of the addressed devices. Furthermore, the application parts must be able to organize themselves (to find and address each other within the network, for instance) to fulfill the desired functionality.

Figure 1a shows the main controller process of our "Music Follow Me" example, which allows music to follow people as they move from room to room in their e-Homes. As individuals enter a new room, the audio receiver will tune to the same radio channel as the receiver in the previous room. Hi-Fi, TV, and PC systems might also act as audio sources, regardless of how they receive audio signals (broadcast, cable, satellite, or Internet stream). Furthermore, we assume that the rooms are equipped with motion detectors. The controller process is event-driven and reacts on two different events. The first event is published by the audio receiver whenever the radio is set to a different channel. The controller process simply stores the new channel in a variable. The second event is detected when a person leaves a room and enters a new one. Then, the controller process tries to find a suitable audio player in the new room and, if successful, sets the radio channel appropriately. Note that the channel is initially set to a default value only on the first execution of the controller process.

Roles

Roles are an abstract concept that was first introduced by Charles Bachmann and Manilal Daya[6] in data modeling and frequently adapted thereafter. We use roles to enable developers to define a certain behavior and specify required properties without determining a concrete entity (device, component, or service, for instance) at design time that has to implement the role and execute it at runtime. Thus, an entity can dynamically adopt a role, execute it for the time it's needed, and abandon it afterwards. Furthermore, a particular entity can perform more than one role at a time, whereas a particular role can also be executed by many entities. Because roles are an abstract concept, how to actually implement a role is open—this can be done as a service, in a separate process, or as an event handler, depending on the granularity of consideration and the target platform's constraints. However, roles are always bound to the particular context in which they interact. They usually exist as long as the context persists, whereas the entities performing the roles might also exist outside this context.

With our approach, roles are extracted from the application model and realize a certain part of the application's functionality. Thus, their context is the application itself. Each role presumes a set of capabilities that describe the minimum requirements for a device to be able to execute that application part. Referring to figure 1, we can divide our example application into three different roles. Role A detects motion and subsequently publishes events when a person enters a new room. This role is expected to run on devices equipped with appropriate sensors. Role C runs on the audio devices, controls the radio channels' settings, and publishes events whenever the user changes the channel. Role B acts as a process controller that receives events published by Role A and Role C and realizes the application's main control flow. It keeps track of the selected radio channel and tunes the new audio source appropriately when the person enters a different room. The process controller role doesn't require any special hardware and any computing device can execute it.

Because roles address only other roles and not a concrete node, they decouple the distributed application from the actual nodes it runs on. Thus, a robust role assignment mechanism is an

important precondition for this role-based development. First, it assigns all roles that are needed to nodes that are capable of serving them. Then, it has to monitor the assigned roles and reassign them if necessary to ensure that all applications are running properly. Because of AS-Nets's dynamic nature, it's important that the mechanism works as autonomously as possible—that is, it automatically adapts to dynamic changes and recovers from transient faults without external intervention. Using a role concept, we can enrich applications at runtime with properties such as self-stabilization or self-organization by simply exploiting the structure of roles defined in the model.

Self-Stabilization

Our approach uses self-stabilization to create robust applications that require minimal manual intervention. Edsger Dijkstra introduced the notion of self-stabilization in his seminal paper published in 1974.[7] A self-stabilizing system is guaranteed to recover from any transient fault within a bounded number of steps provided that no further fault occurs until the system is stable again.[8] The maximum number of steps required to bring the system back into a legitimate state is called *stabilization time*. Self-stabilization can be proven by showing that the system satisfies convergence (started from an arbitrary state, it reaches a legitimate state within a bounded number of steps) and closure (once the system has reached a legitimate state, it stays in the set of legitimate states if no faults occur).[9] In contrast to a self-stabilizing system, a system that isn't self-stabilizing might never reach a legitimate state again once it's corrupted by a fault.

Transient faults include temporary network link failures resulting in message duplication, loss, corruption, insertion, arbitrary sequences of process crashes with subsequent recoveries, and arbitrary perturbations of data structures of any fraction of the processes. Experiencing such faults in AS-Nets isn't unusual due to fragile wireless links and the error-prone nature of resource-constrained devices. However, it's important to note that self-stabilization doesn't mask faults. This is in contrast to other fault tolerance mechanisms that apply redundancy to mask certain faults but don't guarantee convergence if a fault occurs that can't be masked.

In our scenario, we consider convergence guarantees as more fundamental because manual intervention by users must be avoided. If, for example, the device executing the process controller role in the Music Follow Me example fails, the application's self-stabilization property guarantees that the role gets reassigned to another node. This, however, might lead to the selection of a different radio channel when users change rooms as the variable storing the current channel is reinitialized. Although the fault's effects are clearly noticeable and probably inconvenient—the fault hasn't been masked—we believe that users would rather tolerate this behavior than an application that gets stuck and requires personal attendance to get working again. Even worse, it's usually not obvious to users which parts of the application are affected and where the devices that execute them are located.

Self-Organization

One interesting feature of self-stabilizing algorithms is that they don't need any initialization and thus are perfectly suited for systems that must organize themselves. As a result, users might simply put a device featuring a self-stabilizing communication algorithm in place and it will be integrated autonomously into the network. Furthermore, self-stabilizing algorithms can be layered on top of each other to create a self-stabilizing algorithm stack as long as no cyclic dependencies exist among the different layers. This transparent stacking of self-stabilizing layers is a standard technique referred to as *fair composition*.[8] In a previous paper, we analyzed a self-stabilizing role-assignment algorithm stack for AS-Nets that was built on top of a lightweight publish/subscribe system using a self-stabilizing spanning tree algorithm to establish communication routes.[10]All of the basic algorithms must be proved, but once they have, they can be incorporated into the algorithm toolbox and made accessible to model transformers and code generators in a reusable fashion. The self-stabilizing role-assignment algorithm stack has become an integral part of the MODOC project infrastructure that every device is expected to support.

Figure 2 gives an overview of the role assignment algorithm stack that presumes only a simple radio interface able to receive and broadcast messages. The first and lowest layer of the role assignment

algorithm stack structures the network by applying a spanning tree algorithm on the devices that are part of the AS-Net. Thereby, communication routes are established along the edges of the tree to forward messages. Note that it's also possible to support networks with a fixed or heterogeneous infrastructure. In this case, different algorithms which might build on existing technologies such as Jini (www.jini.org) or UPnP (www.upnp.org), must be used in the stack to construct the spanning tree. The second layer comprises a hierarchical publish/subscribe algorithm. In the tree, each device provides a publish/subscribe interface to its children while behaving as a client (publisher or subscriber) when interacting with its parent. Using the interface, devices can subscribe for messages and publish messages related to a certain role instead of having to address these messages directly to the nodes the role runs on. The publish/subscribe algorithm takes care that a message is forwarded to the correct receiver. Furthermore, this indirection also facilitates the migration of roles because publishers are no longer required to know their subscribers. Finally, the third algorithm layer manages role assignment within the network. An elected role coordinator monitors the routing table of the underlying publish/subscribe layer to determine whether there is a subscription for each role—that is, there is a node properly subscribed for each role. If not, a message is sent to a capable candidate node to activate a missing role. Each layer of the algorithm stack leads to a specialization of the node and, thus, contributes essentially to the network's self-organization and its applications. Although every node runs the same spanning tree algorithm on the lowest layer, only some nodes take over specific tasks in higher layers, for example, providing a publish/subscribe interface or acting as role coordinator. These tasks might also be seen as infrastructure-specific roles that every device must be able to perform if required.

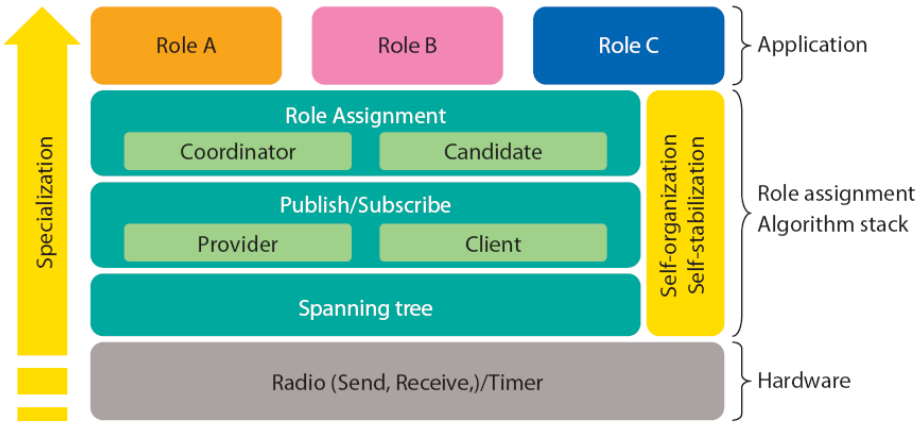

Figure 2. The role assignment algorithm stack employed in the MODOC project.

Leveraging the Role Model at Runtime

We've described the model-driven development process employed in the MODOC project and introduced roles and self-stabilization as fundamental concepts. Obviously, self-stabilization mechanisms have their relevance at runtime (to realize fault tolerance), but in the e-Home scenario we consider in this article, they can't work alone without knowledge about the application roles. This knowledge is derived from the intermediary role model generated in the first transformation step and kept as metainformation during the transformation process. A device must simply know which roles it can perform. Thus, it has to compare its capabilities to the requirements of a particular role that are stored in the metainformation. Using the basic publish/subscribe system, devices subscribe to messages related to roles they might possibly execute and await their activation through an assignment.

A previously chosen node acting as role coordinator is responsible for assigning roles to capable candidates, for monitoring the assigned roles afterwards, and for reassigning them if necessary. To accomplish this task, the coordinator must know which roles belong to a particular application. If at least one candidate exists for every required role, the coordinator performs the assignment to prevent

applications from running partially. For the same reason, the coordinator stops the whole application if a required role can't be reassigned anymore.

The more details of the intermediary role model are preserved as metainformation available to the role coordinator and the devices, the more adequately the role assignment can be carried out at runtime. Several devices at the coordinator's disposal might be able to basically fulfill a certain role. However, some might be better suited than others. Figure 3 shows a Hi-Fi system, an LCD screen with integrated speakers, and a polyphonic smart phone that can all generate audio output (considering the quality for the playback of music, the Hi-Fi system is the best choice). If role requirements also contain metainformation about the desired quality of service, devices can additionally advertise how good they are at performing a particular role. In figure 3, this is depicted by the respective role symbol's size within an implementation, and it can be expressed by a role-specific score value that devices incorporate in their role subscriptions. Based on the score values, the role coordinator subsequently determines the most convenient assignment for users in the present context.

However, this assignment currently doesn't take network limitations into account—that is, two roles that have to communicate heavily might be assigned to different nodes that only share a small-bandwidth link. In future work, we will consider using metainformation to optimize the role placement at runtime to circumvent this. We plan to exploit knowledge derived from the application model as well as from the fragmentation process of an application into roles to mark the roles that are presumed to have a high communication demand. Figure 3a indicates these communication demands by the thickness of the arrows that connect communicating roles. Within an implementation, communication demands can be characterized by latency and bandwidth requirements. Subsequently, we have to revise our role assignment mechanism such that it initially assigns communicating roles to nodes that are located close to each other (in terms of network hops or provided bandwidth of the connecting link). That's why Role B in figure 3b has been assigned to the LCD screen, although the PDA advertises itself to be better suited. However, because the LCD screen also fulfills the QoS-requirements sufficiently, it wins by being close to the sensor node executing Role A with which B is interacting heavily. Thereby, we hope to gain the coarsest performance improvements, providing a good starting point for a more fine-grained optimization afterwards. Nevertheless, such a fine-grained optimization that is carried out dynamically at the application's runtime is still necessary as communication patterns might vary over time.

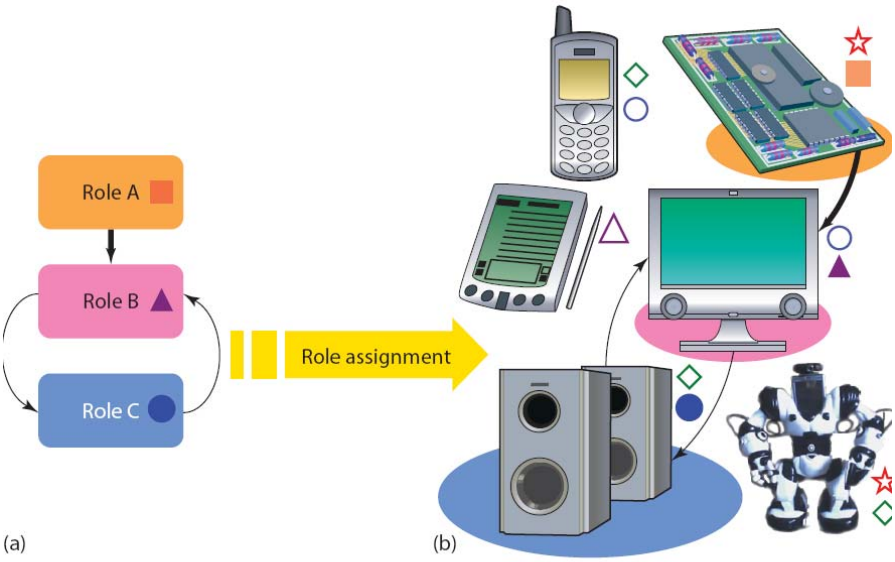

Figure 3. Exploiting (a) QoS-requirements and communication demands stored in the role model to (b) improve the dynamic role assignment at runtime.

Our model-driven approach aims to encapsulate the necessary expert knowledge in the model-transformation process to free application developers from having to care about distribution, heterogeneity, deployment, and self-organization. Models contain knowledge about behavioral and structural aspects of the application and are a necessary prerequisite to be able to automatically guide self-organization processes to their intended goals. Thus, they increase the degrees of freedom for a given application while still keeping it controllable.

Our future work will reverse the transformation process—that is, from running applications back to their models. Specifically, visualizing the application's state and combining it with the original model will aid developers in testing and debugging their applications. Users will benefit because visualizations and combined models enable them to monitor their applications and better understand the self-organizing processes that drive their systems, which will increase the acceptance of autonomous control applications in daily life. By addressing both transformation directions, our work will close the loop and provide a comprehensive model-driven engineering method.

Acknowledgments

We are grateful to the anonymous reviewers for their valuable comments that helped us to improve the article. The work presented is part of the Model-Driven Development of Self-Organizing Control Applications (MODOC) project, which was funded by the German Research Foundation (DFG) priority program 1183 Organic Computing.

References

1. M. Weiser, "The Computer for the 21st Century," *Scientific American*, vol. 265, no. 3, 1991, pp. 94–104.
2. H. Parzyjegla et al., "A Model-Driven Approach to the Development of Autonomous Control Applications," *Proc. 1st Workshop on Model-driven Software Adaptation* (M-ADAPT 07), Berlin Univ.Technology**,** 2007, pp. 25–27.
3. N. Bencomo, G. Blair, and R. France, "Summary of the Workshop Models@run.time at Models 2006," *Models in Software Eng.*, T. Kühne, ed., Springer, 2006, pp. 227–231.
4. T. Weis et al., "Rapid Prototyping for Pervasive Applications," *IEEE Pervasive Computing*, vol. 6, no. 2, 2007, pp. 76–84.
5. A. Phillips and L. Cardelli, "A Correct Abstract Machine for the Stochastic Pi-Calculus," *Proc. Concurrent Models in Molecular Biology* (Bioconcur 04), LNCS 3170, Springer, 2004.
6. C.W. Bachman and M. Daya, "The Role Concept in Data Models," *Proc. 3rd Int'l Conf. Very Large Data Bases*, IEEE CS Press, 1977, pp. 464–476.
7. E.W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Comm. ACM*, 1974, vol. 17, no. 11, pp. 643–644.
8. S. Dolev, *Self-Stabilization*, MIT Press, 2000.
9. A. Arora and M.G. Gouda, "Closure and Convergence: A Foundation of Fault-Tolerant Computing," *Software Eng.*, vol. 19, no. 11, 1993, pp. 1015–1027.
10. T. Weis et al., "Self-Organizing and Self-Stabilizing Role Assignment in Sensor/Actuator Networks," *Proc. 8th Int'l Symp. Distributed Objects and Applications* (DOA 06), R. Meersman and Z. Tari, eds., LNCS 4276, Springer, pp. 1807–1824.

Helge Parzyjegla is a PhD student at the Communication and Operating Systems group at Berlin University of Technology. His research interests include self-organization principles for ubiquitous computing, event-based systems, publish/subscribe middleware, and peer-to-peer networks. He has an MA in computer science from the Berlin University of Technology and is a member of the ACM. Contact him at parzyjegla@acm.org.

Michael A. Jaeger is a postdoctoral researcher at the Communication and Operating Systems group at Berlin University of Technology. His research interests include event-based systems, publish/subscribe middleware, self-management, and self-stabilization. He has an MA in computer science from the Goethe University of Frankfurt/Main, Germany. Jaeger is a member of the ACM and the German Computer Science Society (GI). Contact him at michael.jaeger@acm.org.

Gero Mühl is a postdoctoral researcher at the Berlin University of Technology. His research interests include distributed systems, middleware, event-based systems, self-organization, and ubiquitous computing. He has a PhD in computer science from the Darmstadt University of Technology. He is a member of the ACM and the German Computer Science Society (GI). Contact him at g_muehl@acm.org.

Torben Weis leads the Distributed Systems Department at the University Duisburg-Essen. His research interests are in model-driven computing, pervasive applications, and peer-to-peer systems. He has a PhD in computer science from the Berlin University of Technology. Contact him at torben.weis@uni-due.de.

Cite this article:
Helge Parzyjegla, Michael A. Jaeger, Gero Mühl, and Torben Weis, "Model-driven Development and Adaptation of Autonomous Control Applications," *IEEE Distributed Systems Online*, vol. 9, no. 11, art. no. 0811-oy002.