Model-Driven Development of Ubiquitous Applications for Sensor-Actuator-Networks with Abstract State Machines

Sebastian Schuster and Uwe Brinkschulte

Institute for Process Control and Robotics,
Universität Karlsruhe(TH), Kaiserstraße 12, 76128 Karlsruhe
sschu|brinks@ira.uka.de

Abstract. The development of applications in the domain of Ubiquitous Computing has to deal with some unique challenges. The target environment consists of very heterogeneous and partly low-power devices. It changes rapidly due to wireless communication and mobile users. We propose to use model-driven development based on Abstract State Machines to deal with these challenges. Applications are defined on high levels of abstraction and efficient implementations tailored to the target platform are automatically generated.

1 Introduction

Mark Weiser's vision of Ubiquitous Computing (UC) [9] describes a world where computers are everywhere and support your everyday life. They relieve you from routine work, which makes UC attractive to many people, as it does to us.

Today, users have to tell the computer what to do and enter information in the way the machine wants it - the main issue Weiser had with the way we use computers. To realize UC, computers and other devices must be enhanced to detect the user's needs and to support him actively.

Wireless sensor networks (WSNs) [3] consist of simple, low power, and cheap sensor nodes, working together to monitor their environment. Thus, they can serve the purpose of detecting the user's actions. By adding nodes with the capabilities to influence their environment, a sensor and actuator network (SAN) can be established and serve as an infrastructure for ubiquitous applications.

SANs will include all kinds of devices from different vendors, ranging from full-featured PCs over PDAs and Smartphones to tiny, low-power sensor nodes and embedded devices tailored to specific needs. Some of these nodes are stationary, and some will be mobile. Different kinds of applications are possible: there will be applications bound to a specific environment, like controlling the lighting based on user presence. Other applications will be bound to a specific user and will control the environment based on the user's preferences, like controlling TV, heating or air condition. Some applications will mainly provide information, e.g. cooking recipes or traffic guidance. All of these applications will run simultaneously and have to share resources. They must possibly interoperate without

knowing each other. They must adapt themselves in an ever-changing environment, from switching input and output devices when the user moves to showing a very different behavior depending on the current context.

At a first glance, using established development techniques from traditional distributed systems for SANs, like middleware, may seem to be a good idea. For a number of reasons discussed in section 2, this is not feasible for SANs. However, without powerful tools raising software development productivity of ubiquitous applications, there is no chance that Weiser's vision will ever be realized. Ubiquitous applications will stay a toy for the wealthy people instead.

After introducing the major challenges in the area of ubiquitous application development in section 2, we discuss related development tools tailored to ubiquitous applications in section 3. Afterwards, we present our arguments for model driven techniques in this application field in section 4. Furthermore, we sketch our approach to realize a model-driven development process. The work of implementing this approach is in progress. We are optimistic our ideas will prove to be valuable in practice. This paper concludes with a summary in section 5.

2 Challenges

There are a number of challenges to be addressed when developing ubiquitous applications. Obviously, a ubiquitous application is a distributed one. Multiple processes run within the SAN and communicate by exchanging messages. Typical challenges of distributed applications include partial failures, transmission errors, and synchronization. All of these are well researched. Furthermore, solutions to deal with these problems are incorporated in middleware, ready to be reused by the developer. However, what are the challenges that do not allow to transfer existing solutions to the domain of Ubiquitous Computing?

2.1 Efficiency

Since nothing comes for free, the advantages of using a middleware introduce costs. The computation steps done in the middleware consume time and energy, while the necessary code takes memory space and energy. The resources of sensor nodes in computation power, memory space, and energy, are very constrained. This, putting an upper bound on the amount of work that can be done on a sensor node, becomes a challenge of efficiency when using a middleware.

Middleware is supposed to offer flexible solutions to a diverse range of applications. The tailoring to the needs of the application happens mostly at runtime, e.g. when the application feeds parameters to middleware function calls. Selecting the proper middleware functionality according to these parameters takes extra computation steps. Furthermore, many functions are unused, despite taking memory space. One can generally say – with a classical middleware – higher flexibility decreases efficiency (while facilitating reuse). How to deal with this tradeoff for SANs is an open question.

2.2 Heterogeneity and Interoperability

A typical task of middleware is to deal with a heterogeneous system consisting of nodes with different properties. Its goal is to hide the differences from the programmer and make the system look like a homogeneous one, easing software development. When the nodes of the network are not too different in terms of processing power and storage space, this can be achieved by including standard communication protocols, conversion of different data representations etc. within the middleware. In a system with nodes ranging from tiny sensor nodes to full-featured personal computers, with resources differing by orders of magnitudes, this is nearly impossible. However, in the absence of powerful abstractions, programmers would have to write specific code for every kind of node, manually adding functions to make the nodes interoperate. It means resolving problems already solved for traditional distributed systems – surely not the best way to go.

2.3 Dynamics

Traditional distributed applications often assume to run upon a fixed network. Processes communicate directly and reliably – the developer does not see details like network routing or location information. A node unreachable for some reason is treated as a failure and handled by the middleware or the application. However, in ubiquitous environments, users carry nodes around, nodes use unreliable wireless communication, their energy can be exhausted, and the user can interfere with the system in unforeseen ways. Communication failures are common and network connectivity changes rapidly. Since the user should not be bothered to deal with exceptions, self-organizing algorithms that make the system adapt itself to changes autonomously are necessary. These algorithms should be generic and flexible enough to make them available for reuse for a wide range of ubiquitous applications. At the same time, efficiency must be preserved.

2.4 Goals

For a productive development of ubiquitous applications, solutions for efficiently dealing with heterogeneity and the dynamics of the system must be available for reuse. The developer should describe system behavior on a high level of abstraction, hiding differences between nodes and network changes. Applications cannot be custom made for each environment – this would be much too expensive. The developer might not even know the system his application has to run on. Specifying in abstract terms that can be found in any ubiquitous environment is the only way possible. Instead of specifying on the level of individual nodes, stating *node X turn the light on*, the developer must be able to code an equivalent of *turn on the light in the user's room*. Detecting the presence of the user in a room and finding a node with a certain capability – like turning on the light – is something that will happen regularly in ubiquitous applications. Implementing

these functions adaptable to different environments once and reusing them is a prerequisite for high development productivity.

However, the target application must not only be adaptable to different environments. People may have different requirements regarding privacy issues or they want their daily life support to be a little different. Applications must be customizable to the varying needs of the users.

3 Related Work

Since the research area of Ubiquitous Computing is quite young, most of the work has been carried out in trying to solve certain problems and not in making these solutions available for reuse. However, two proposals explicitly dealing with some of the identified challenges had a major influence on our work.

The first one is PCOM [8], a component-oriented middleware for pervasive applications. PCOM applications consist of a tree of components, each implementing parts of the application functionality. The actual layout of tree instances is determined by the PCOM middleware at runtime – based on capabilities of the different nodes and requirements of each component given by the developer in some XML-dialect. Thus, the application can also be adapted to changes in the environment. Motivated by the *development-by-composition*-paradigm, components implemented once can be reused in other applications as well. PCOM is built upon another middleware layer, BASE [1], offering communication services in heterogeneous and dynamic environments, relieving the developer from dealing with network routing. BASE and PCOM transfer the traditional approach – using layers of middleware – to the development of ubiquitous applications, explicitely considering highly dynamic and heterogeneous environments. Their memory footprint is about 120-160KB, preventing to use them on sensor nodes. The level of abstraction that can be achieved depends on the available components. Specifying tree composition in some XML-dialect and using general purpose programming languages to implement components without further support is still way off developing applications in terms of the target domain.

An approach motivated by the OMG's Model Driven Architecture (MDA) is described in [7]. The OMG proposes to use models and not code as the primary artifact of software development. While models are widely used in software development, serving as a sketch for the real code, they tend to get out of synch as code development evolves. The resulting code is always a mix of parts dealing with the real business problems and, to a large amount, of parts due to the way these problems have to be solved on a specific platform. Problem domain and realization domain should be clearly separated instead, by describing the application in platform-independent models (PIM) containing application logic only. Afterwards, they are transformed to platform-specific models, enriched with platform details. In the last step, executable code can be generated. These transformations can be performed manually or automatically – the latter one being the preferred way.

In [7] a language with a fancy graphical representation to describe platform-independent models of applications for home automation is defined. The developer describes the behavior of the system using several communicating state machines running in the system. When deploying the application in a target environment, the state machines of the PIM are split up into roles, which can be assigned to nodes of the target system. These roles are transformed to executable code and installed on the target nodes, depending on their capabilities. Our lighting application would consist of two roles, one for detecting the presence of a user, and one for turning on the light. The first one would be installed on all nodes with a motion detector, the second on all connected to the lighting. The system can adapt to changes by activating and deactivating nodes, e.g. roles of failing nodes can be taken over by others. Multiple platforms can be supported by developing the necessary transformers and code generators. The generative model-driven approach allows to generate efficient code specifically tailored to the target nodes, avoiding the overhead of a middleware. At the same time, the roles-based approach can deal with a dynamic system, suggesting a way to deal with the flexibility vs. efficiency tradeoff in classical middleware. Our approach is based on this idea too. The described development method lacks ways of ensuring interoperability. On the highest level, descriptions based on finite state machines can probably be improved with terms more closely resembling the domain of Ubiquitous Computing.

4 Proposed Solution

We propose to use model-driven development to handle the identified challenges. The aim is to combine the advantages of using a middleware, development on a high level, with the generation of code tailored to different platforms for higher efficiency. The functionality provided by a middleware is added by model transformations instead. At the same time, the overhead introduced by a middleware is avoided. Applications can be developed in a coherent way for heterogeneous target environments that include devices as resource-constrained as sensor nodes. When installing an application, it is transformed automatically, taking user preferences and properties of the target environment into account.

While realizing this vision will surely be appreciated, a lot of work lies ahead. The main questions that have to be answered include: How do the models look like? How to define transformations? How to guide them? We present first answers to these questions in the following sections. We are currently at the start of developing and implementing our development process, following a bottom-up approach. Our concept certainly needs further refinements.

4.1 Process Overview

Different kinds of models are involved in the development process: models describing user preferences, models describing the target environment, and models describing the behavior of some entity. At the top-level, the developer implements

the application by specifying a platform-independent behavior model, describing how the environment reacts to what the user is doing. On lower levels, the behavior of parts of the system down to individual nodes is specified. Compared to the MDA, we propose to use multiple transformation steps from top-level models to executable code. The available transformations are arranged in a hierarchy, each transformation bridging a smaller gap. Models are transformed along the edges, starting at the highest level of the hierarchy and yielding executable code at the leaves. The direction to take when traversing and how to transform is controlled by the target environment and user preferences – the available devices decide which transformations to take. Wether to add encryption algorithms depends on the user preferences for example.

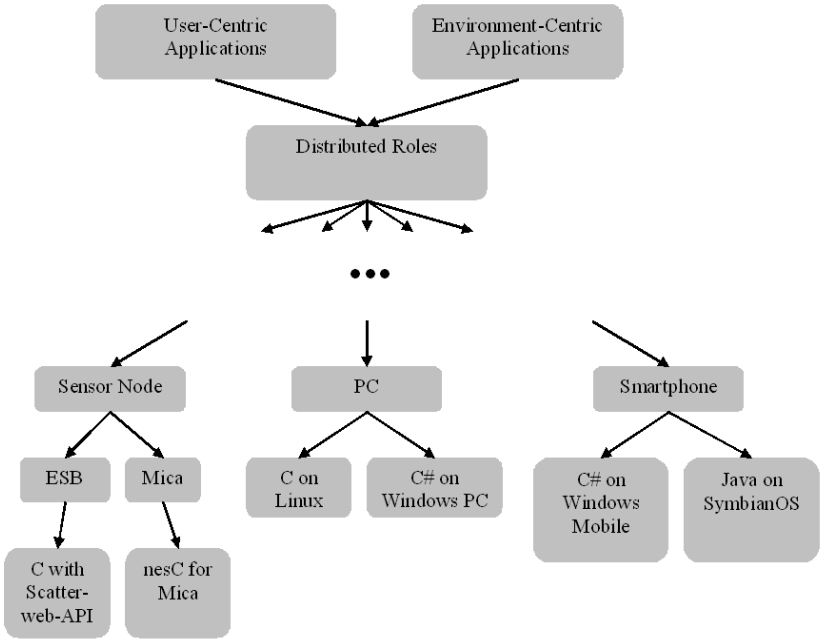

Fig. 1. Transformation Hierarchy.

An extract of the transformation hierarchy is given in figure 1. On the highest level, different modeling languages can be used to describe different kinds of applications. These can be transformed to a language where application functionality is decomposed into distributed roles, dealing with the dynamics of the environment – similar to [7]. Several transformations not shown here add communication and interoperability support or customizations. After that, transfor-

mations generate models for different platforms, containing the roles the target node can take. At first, these models will be generated for generic platforms like *Sensor Node* or *PC*, using features offered by all types of sensor nodes or PCs respectively. The generic models are then transformed to models for specific device types, like Mica or Scatterweb ESB sensor nodes. Eventually, executable code can be generated.

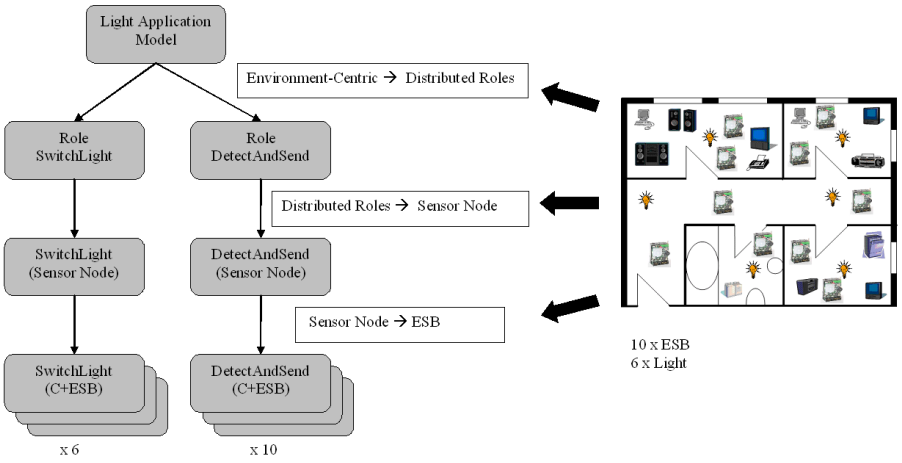

Fig. 2. Example Transformation.

An example transformation process is given in figure 2, showing the installation of an application controlling the lighting based on user presence. The transformation processor (a device with less resource constraints like a PC) recognizes ten Scatterweb ESB [6] sensor nodes able to detect user presence. Six of the nodes can control a light. When transforming to the next role-based modeling layer, it deduces there will be two types of roles necessary. The first (*DetectAndSend)* detects user presence and informs the second role (*SwitchLight*) able to control the light. These roles are then converted into models for generic sensor nodes, describing how nodes communicate and activate their roles. In the last step, C-code is generated from these descriptions and flashed onto the nodes.

Composing the transformation chain of smaller transformations facilitates their reuse. Introducing a new type of sensor node only needs a less expensive transformation from the generic platform to the new platform for example. The role-based decomposition of application functionality allows to adapt at runtime. By adding additional roles, e.g. for data conversion, interoperability can be assured. Finally, the generated code is more efficient than a middleware-based approach, leaving out unnecessary features.

4.2 Abstract State Machines as Behavior Models

Models are defined in terms of a modeling language, describing what models look like (syntax) and how to interpret those (semantics). At the top-level, we need expressive languages specific to Ubiquitous Computing. Since ubiquitous applications let the environment support the user, reacting to what the user is doing, this language will feature an event-driven control flow. A language suitable for home automation would offer terms like *Room*, *Lighting* or *TV*. At lower levels, we need languages to describe roles and the behavior of nodes.

When transforming models, we have to make sure that the resulting model describes a behavior equivalent to the source model. A top-level model that includes an abstract action *Alert the user*, may be correctly transformed to a ringing Smartphone or a message shown on screen of a TV – depending on the target environment and the current situation. Turning on the washing machine is most likely not an equivalent action. While ambiguity can be intended as a consequence of abstract specifications on a higher level, unwanted ambiguity must be avoided. The key is a precise – formally specified – semantics of the modeling languages. Many errors made in software development are due to informal natural language specifications, interpreted differently by different people working on the same project. In a multi-step transformation as we propose it, using formally specified languages is even more important.

Abstract state machines (ASMs) [4] can be used to formally describe every algorithm on any level of abstraction. Formally specifying behaviors on different levels of abstraction is exactly what we need, making ASMs an appealing candidate to be used on the different levels of our multi-step transformation.

ASM structure An ASM consists of two parts: the description of the state of the machine and a set of rules governing the transitions from one state to the next. The state is described in terms of an algebra – sets with operations and relations. The author of [5] argued that *...every static mathematic reality can be described as a structure in the sense of mathematical logic....* The rules are made up of conditions guarding the firing of the rule and of updates describing how to change the state of the machine. Starting in an initial state, the machine performs step by step, in each step executing all matching rules and updating the state of the machine in one atomic step.

ASMs for Ubiquitous Applications The following example shows an excerpt from a high level ASM describing our lighting application.

```
enum Rooms = {Livingroom, Bathroom, Kitchen, Bedroom}
function Light: Rooms -> BOOLEAN
function Occupied: Rooms -> BOOLEAN

rule Main = par
forall room in Rooms do Light(room):=Occupied(room) endforall
endpar
```

A set *Rooms* is defined consisting of the different rooms in the ubiquitous environment. The function *Light* can be used to control the lighting in every room. A function *Occupied* returns true if anybody is in a room. The only rule *Main* states, the lights should be switched according to user presence.

This is a description on a very high level. It contains what could be called the business logic of the application. Since ubiquitous applications are about relieving people from routine tasks, we argue that their business logic is not too complex and compact descriptions on a high level are possible. ASMs can also be used on a lower level to describe the behavior of a role or of a single node. A sensor node can be described in terms of the state of its sensors and functions yielding current sensor values.

In order to use ASMs for behavior descriptions, a vocabulary to describe the ASM states has to be defined at the different levels. At the highest level, there will be sets and functions like the ones shown above. For lower levels, functions showing the state of sensors have to be defined for example. We are currently investigating possible vocabularies for sensor nodes.

Transforming ASMs The vocabularies of the different levels essentially describe our modeling languages – the terms that can be used to describe the state of system with ASM rules describing the behavior of its entities. How to define the transformations, mapping the abstract function *Occupied* to ASMs describing roles, that observe the motion detection sensor and transmit a message when movements are detected?

The ASM method [2] describes a software development process based on ASMs. According to this method, a developer starts with a high-level ASM describing the application under construction. This ASM is refined stepwise – gradually enriching it with details describing how to implement what was specified on the higher level. Functions can be replaced by additional ASMs computing this function or ASMs can be composed of sub-ASMs. Sequential ASMs can be refined by adding agents, making it a distributed ASM. All of these steps have to be performed manually by the developer. What we need is a way of automating these steps - this would eventually yield our transformation chain.

To establish our approach in practice, an expressive specialized language to describe ASM transformations would be necessary. We are currently considering general purpose languages using established model transformation and code generation patterns only. We first want to investigate how to apply transformations and how to parametrize them. The application of special transformation languages is planned for the future.

5 Summary

We proposed to use model-driven development techniques to deal with the primary challenges of developing ubiquitous applications: high degrees of heterogeneity, the need for efficiency, and high dynamism in ubiquitous environments.

Applications are described as high level models independent of a specific platform and are automatically transformed to platform-specific models matching the target environment. The core idea of our approach is to use a chain of transformations with small transformation steps. Building new transformations and including new platforms will be less expensive and reuse is facilitated.

Introducing multiple transformation steps, unwanted ambiguity through specifications given in languages with informally defined semantics becomes an even bigger problem. Therefore, we proposed to use Abstract State Machines to formally describe the behavior on all levels of abstraction. We sketched how ASMs can be used on different levels and why they are suitable for transformations.

The next step will be define vocabularies for ASMs on the different levels. Afterwards we will investigate how transformations can be defined. We plan to implement a complete transformation chain based on ASMs – including decomposition of application functionality and adapting the composition at runtime.

6 Acknowledgements

Sebastian Schuster is supported by the German Research Foundation (DFG) within the Research Training Group GRK 1194 Self-organizing Sensor-Actuator-Networks.

References

1. Christian Becker, Gregor Schiele, Holger Gubbels, and Kurt Rothermel. BASE - A micro-broker-based middleware for pervasive computing. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communication (PerCom) USA*, pages 443–451. Los Alamitos: IEEE Computer Society, March 2003.
2. E. (Egon) Börger and Robert F. Stärk. *Abstract state machines: a method for high-level system design and analysis*. Springer-Verlag, 2003.
3. D. Estrin, G. Pottie, L. Girod, and M. Srivastava. Instrumenting the world with wireless sensor networks. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2001)*, June 2001.
4. Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.
5. Yuri Gurevich. Abstract state machines: An overview of the project. In Dietmar Seipel and Jose Maria Turull Torres, editors, *FoIKS*, volume 2942 of *Lecture Notes in Computer Science*, pages 6–13. Springer, 2004.
6. Jochen H. Schiller, Achim Liers, Hartmut Ritter, Rolf Winter, and Thiemo Voigt. Scatterweb - low power sensor nodes and energy aware routing. In *HICSS*, 2005.
7. Andreas Ulbrich, Torben Weis, Gero Mühl, and Kurt Geihs. Application development for actuator- and sensor-networks. In *4. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze*, Zurich, Switzerland, March 2005.
8. Torben Weis, Marcus Handte, Mirko Knoll, and Christian Becker. Customizable pervasive applications. In *PERCOM '06: Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM'06)*, pages 239–244, Washington, DC, USA, 2006. IEEE Computer Society.
9. Mark Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94 – 104, September 1991.