Generating a Tailored Middleware for Wireless Sensor Network Applications

Christian Buckl, Stephan Sommer, Andreas Scholz, Alois Knoll, Alfons Kemper
Department of Informatics
Technische Universität München
Garching b. München, Germany
{buckl,sommerst,scholza,knoll,kemper}@in.tum.de

Abstract

Wireless sensor networks are characterized by resource constraints. Therefore, today's sensor networks are implemented from scratch emphasizing code efficiency. This development strategy leads to relatively complex code and bad code reusability in further projects. To improve reusability and development efficiency, it is state-of-the-art in the development of standard information systems to divide applications into at least two parts, the application-logic, providing all the functions to solve a given problem and a reusable distributed middleware providing a container for the application. After developing the middleware once, the developer of further projects need to focus only on the application-logic. Thereby, the development times can be reduced considerably. However, a generic middleware layer replacing code implemented from scratch is not practicable in sensor networks due to resource constraints.
Within this paper, we will present a model-driven approach in combination with a template-based code generator to get the best of both development strategies. This approach enables us to generate a tailored middleware for our application including interface-stubs for the application-logic. In contrast to other component-based approaches, the templates can be adopted easily to fulfill specific platform needs. We will demonstrate the practicability of this approach by implementing the control of a model railway.

1 Introduction

Developing wireless sensor network applications requires a different approach than developing standard information systems. Many problems such as mobility, limited resources and unreliable communication links, must be considered by the developer. In most other domains, these problems are solved by underlying abstraction layers. In standard information systems, these abstraction layers are realized by a middleware that offers high-level services to the developer. But due to the limited resources available on wireless sensor nodes, a generic middleware providing a container for all applications in the sensor network is quite impossible. Although, one can currently observe the advent of new, more powerful nodes such as iMote2 [3] that enable the use of a generic middleware, e.g. the .net Framework [17], these nodes are too expensive and require too much power to be used in many sensor network applications.
Resource constraints such as available main memory or limited power supply force the developer to implement these services, typically provided by middleware, manually and tailored for a specific hardware and application. Therefore, developers with expert platform and low-level programming knowledge are required. This application-specific development of standard functionality is a very slow and complex process which impacts the development time of the whole project and reduces the code reusability. In addition, the resulting code mixes typically aspects concerning application and system logic. Because of this, minor changes in application-logic may lead to vast changes of the whole system; code reuse is quite impossible.
A solution of this problem is an application and platform specific middleware with defined interfaces for applications. The separation of system and application logic helps to split the software development into different parts. Figure 1 shows a logical view of such a component/container infrastructure [21]. A middleware/container offers different services, like component discovery and inter-component communication, to application components denoted by an A in the figure. Hardware related software components, denoted by an H, realize the hardware access (sensing or actuating) and hide implementation details. An application domain expert is able to develop the application-logic without considering each platform detail like communication or sensor-access. On the other hand, developers with low-level programming skills and expert hardware knowledge can develop the services provided by the middleware and used by the application-logic. However, it is important to guarantee that the container realizes only the functionality required by

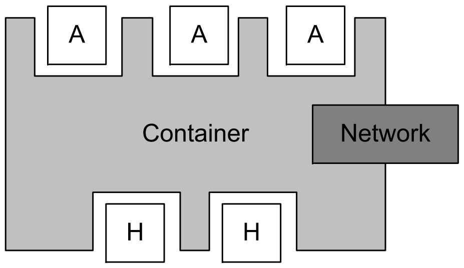

Figure 1. Component/Container Infrastructure

the application and implicates no or only minimal overhead in comparison to the manual implementation.
In this paper, we will present a tool-supported approach that realizes such a tailored middleware. To increase development efficiency, the manual tailoring of its components is replaced by a model-driven development approach. Thus, the approach combines the advantages of component-based and model-based development, as discussed in [19]. The domain expert creates a model of the intended application based on a meta-model describing the relevant features of sensor network applications. Based on the model, a template-based code generator produces the middleware including interfaces for the application-logic. Templates can be seen as highly adaptable components. Each template can solve a particular aspect of the middleware or can be used to construct the middleware out of other templates. Using this approach, we can create a tailored middleware providing exactly the features required by the application. Therefore, we get a good tradeoff between resource-use, code size and development time.
The paper starts with a discussion of the components of a middleware that are useful for wireless sensor networks in Section 2. In Section 3, we describe the models used for the code generation. The code generation technique is explained in Section 4. Section 5 discusses our first prototype. In Section 6, the experiences with the tools are described in the context of a simple application. The related work is discussed in Section 7. Finally, Section 8 concludes the paper and points out possibilities for future work.

2 Middleware

Within this section, we will discuss the tasks and related components of the middleware. The general architecture is depicted in Figure 2. Similar to CORBA [14], we provide well defined interfaces for the application components to access the container-services. But in contrast to CORBA, our container can be tailored for a specific application and hardware.
We expect the system to be heterogeneous concerning the computational power and memory capabilities. Therefore, the nodes can take over different roles within the whole system. Resource-constrained nodes can be used to perform simple interactions with the environment like sensing or actuating. More powerful nodes can control the whole network, optimize the data flow and trigger application changes.
The middleware forms a container that allows an easy combination of the components realizing the application functionality. Regarding these components, we distinguish two kinds of components. An *Application Component* realizes a control function of the application. The functionality can be implemented independently of the underlying hardware. Therefore, these components can be placed within the distributed system according to some performance criteria. In contrast, a *Hardware Interaction Component* realizes the hardware access, e.g. sensing or actuating, and must be implemented hardware dependent.
The middleware realizes the interaction of these components and can be seen as intelligent glue code. In contrast to the operating systems such as TinyOS [18] or SOS [16], which are very often considered to be middleware themselves, the presented approach has to be seen at a higher level. In particular, it offers services related to the distributed execution of sensor applications such as routing, node failure management and quality of service. It consists of several components as depicted in Figure 2. The communication between the different nodes is handled by the *Network Service*. In this service, all supported communication-media, e.g. ZigBee or serial communication, of a single node are implemented. Details about the network protocol and routing are hidden by this service.
Received messages are forwarded to the *Broker Service*. This service handles all communication at the level of application logic. This comprises the local communication between different application components and / or hardware services, as well as the transmission of local results to external components. Every time a new message is received by the network layer or sent by a local component, the Broker Service determines the set of target components. If a target component is located on a different node, the Broker Service sends a message, including the target-node id, to the Network Service.
The configuration of the Broker Service is handled by the *Application Management*. While the Network and Broker Service must be implemented on each node, the Application Management may be realized only on more powerful nodes. The Application Management is responsible for the configuration of the whole application and tries to optimize this configuration according to different criteria, like QoS or maximum life time. The Application Management is supported by a *Component Management* that manages all available application components, as well as a *Node Man-*

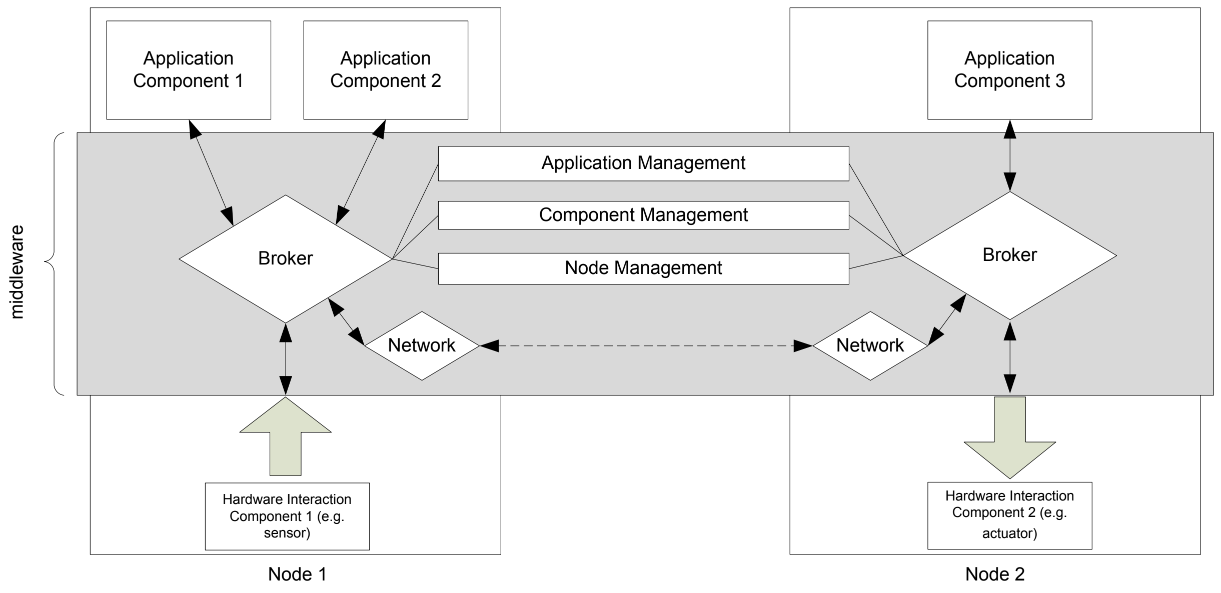

Figure 2. Middleware Services

agement that monitors the set of operational nodes. The task of the Node Management comprises amongst others the discovery of new nodes and the monitoring (e.g. battery status) of connected nodes. The Component Management can be placed similar to the Application Management only on a subset of the available nodes.
After this short overview of all used middleware components, we will discuss some of them in more detail.

2.1 Node Management

The first middleware component, we will discuss in more detail is the Node Management. This distributed service is used to collect status information and capabilities of all nodes in the sensor network. The capabilities of the network comprise the available sensors and actuators, the provided communication media as well as processing power and storage capabilities. In addition, run-time data like battery status, free memory or hardware failures must be monitored. This information can be used to optimize the configuration of the application. Furthermore, the status information can be used for maintenance to identify nodes with heavy load or low energy resource at an early stage and to make arrangements to replace these nodes or their battery.
To gather all these information, it is essential for the whole system that each node announces its presence and keeps the state up to date. A node failure can be detected and reported by neighbor nodes due to the fact that communication to a lost node is not possible anymore. For resource-constrained nodes in the network, a passive version of the node management is sufficient. It is passive in the sense that they provide information about the hosting node but do not collect information about other nodes. More powerful nodes execute active versions of the node management that gather the forwarded information and report changes to the Application Management.

2.2 Component Management

The Component Management provides information about all components available for the entire sensor network. We differentiate between *Application Components* and *Hardware Interaction Component*. Hardware Interaction Components are offered on each node with dedicated hardware devices. In contrast, it is possible to locate Application Components on an arbitrary node in the network. To acquire an optimal service placement in the sensor network, the Application Management service needs in-depth knowledge of all interfaces, the provided functionality and resources requirements (memory consumption, required processor time) of each component. This information is stored, maintained and provided by the Component Management. Different application components may realize a similar functionality. Based on the description of these components, the Application Manager can choose an adequate component based on the available devices and QoS constraints.

2.3 Application Management

This middleware component handles the configuration of the application. The configuration depends on the set of available nodes and their status, the set of software components, the topology and QoS requirements. Application components can be placed intelligently within the distributed system to minimize network load or to balance the load on the different processors. If for example an average value out of a set of redundant sensor results is used at a remote controller, the application component computing this average value should be placed close to the sensors. A new configuration can be obtained by moving the affected software components and updating the routing. The latter is done by reconfiguring the Broker.

2.4 Broker

The component realizing the Broker must be implemented as a local service on each node. The task of this component is to realize the routing at the level of application logic. The routing table of the broker is maintained by the Application Management to guarantee an optimal routing. All messages consumed and/or produced on a specific node need to pass the broker. It is the task of the broker to decide to which components on which nodes the message will be forwarded. In contrast to messages for local services, the messages for non-local services need to be sent over network. The message including routing information such as the receiver, security and reliability requirements is sent to the Network service for further processing.

2.5 Network Service

The Network Service is used to communicate with other nodes in the sensor network regardless of the concrete communication medium. In order to get better efficiency and less overhead, we adapt the capabilities of this service while generating the middleware. With adequate hardware and application knowledge, we can decide which communication-medium is available on a specific platform and which ones of them are actually needed or used for the specific application. This leads to a quite optimal performance without the need of any manual adaptations. The Network Service implements the end-to-end routing by forwarding the message to the next neighbor on the route and applying the appropriate communication protocol such as ZigBee. To achieve secure communication, message de- and encryption can be activated in this service to transparently get a secure communication layer for message transport. For better efficiency and because of the resource constraints, only critical messages are encrypted. Which messages are assumed to be critical, can be derived by the application model that is described in the next section.

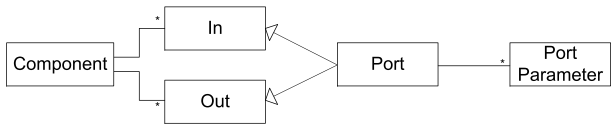

Figure 3. Component Model

3 Domain Specific Language

This section will give an overview of our modeling language used for automatic code generation. To allow an extensive code generation, the modeling language must be *generative* and *descriptive* [10]. The models must have explicit execution semantics, hardware characteristics need to be specified and the interfaces of the components must be described in an unambiguous way. Especially the first requirement excludes the use of standard modeling languages. The widely used Unified Modeling Language UML lacks the precision and rigor needed for code generation [9] for example. Therefore, we decided to design an own domain specific language that is optimized for the use in our specific scenario. Thus, it is possible to create a very simple, but powerful modeling language.
Since it is necessary to describe different aspects of the system, we decided to use several sub-models. The hardware model describes the properties of the hardware, the component model describes the interfaces and parameters of the application components and the application model is used for the specification of the concrete component interplay.
All the meta-models are specified using the Eclipse Modeling Framework[1] (EMF). Several plug-ins for Eclipse are available to specify the models. In the following subsections, we will summarize the used models. Due to space limitation, we will restrict the description on the characteristics that are necessary for the code generation.

3.1 Hardware Model

The hardware model is used to specify the properties of the used hardware. The main idea for this model is to adapt the code generation to the specific platform. In addition, the model is used for optimization issues. Within the model, platform specialists can describe essential hardware features like computing power, available memory and supported communication protocols.

3.2 Component Model

The component developer can specify the component interfaces within this model. Each interface is described as a set of in and out ports. Each in and out port can consist

[1]http://www.eclipse.org/modeling/emf/

```
≪FOREACH app.componentInstance AS ci≫≪IF ci.node==n≫
  Main.StdControl ->≪ci.name≫C.StdControl;
  BrokerC.≪ci.name≫ ->≪ci.name≫C;
  ≪ENDIF-≫
≪ENDFOREACH-≫
```

Figure 4. Template

```
Main.StdControl ->OnOffLEDC.StdControl;
BrokerC.OnOffLED ->OnOffLEDC;

Main.StdControl ->LightClapServiceC.StdControl;
BrokerC.LightClapService ->LightClapServiceC;

Main.StdControl ->SoundSensorC.StdControl;
BrokerC.SoundSensor ->SoundSensorC;
```

Figure 5. Generated Code

of different parameters/variables. The description of the in and out ports is used for the interaction between the different components. We use an event based push model for component interaction similar to data flow diagrams. The activation of an out port is realized by sending a message. This message contains elements for each parameter of the individual out port. The arrival of a message at a specific component triggers the activation of the according in port. Figure 3 shows a simplified version of our meta-model.

3.3 Application Model

The interaction between the different components is specified within the application model. This step comprises a simple wiring of out ports to in ports. Depending on the middleware used, the user also has to specify the mapping of the application components to a specific node or state criteria used for run-time optimization.

4 Code Generation

One key requirement was the application-specific tailoring of the middleware. We are using a template-based code generator [1] to satisfy this requirement. Templates are highly adaptable components. This offers not only the possibility to adjust some parameters of the template, but also to generate strongly application dependent components of the middleware like a routing table of the broker.
Templates can be used to solve certain aspects of the runtime system, or to combine the results of different templates to form the middleware. Most templates are platform dependent in the sense that they offer a solution only for a certain combination of hardware and operating system. Therefore, also the correct selection of adequate templates is necessary.
Instead of implementing an own code generator, we are using an existing code generation framework, called openArchitectureWare[2] [20]. OpenArchitectureWare provides for these problems a special template language, call *XPand*. XPand offers the statements *DEFINE* to declare a new code generation function and *EXPAND* to call other generation functions during the code generation. OpenArchitectureWare also allows polymorphism as one element to select adequate templates.
To specify the control flow of the code generation, the commands *FOR/FOREACH* and *IF/ELSE* can be used. The *FOREACH* statement is used to generate code for each object of a certain type that is declared within the model. Finally, the commands *FILE* and *ENDFILE* allow the management of the generated files. The code generation process is then rather simple: the adaptation of the templates to the model is performed using a technique similar to preprocessor macros. Text sequences between the different XPand commands are directly copied to the generated files and variables allow the access to objects and their attributes.
Figure 4 shows a simple template that illustrates the basic concept. The template realizes the generation of links between the components on one node and its Broker in TinyOS 1.x. The required information can be retrieved from the model. The generated code is depicted in Figure 5.

5 First Prototype

Within a first prototype, we have implemented the main features of the approach discussed before. Using our model-driven development tool, the developer can specify the components of the application. The tool supports the automatic generation of an optimized middleware and integrates the application components. The current prototype is based on a static setting of the individual nodes with all nodes in 1-hop distance. The main task of the middleware is to realize the interaction between the different components on one node and between local and remote components. Therefore, only the Broker and the Network Service are necessary within the middleware.
The logical connection for a simple example is depicted in Figure 6. The application is executed on the two nodes. A Hardware Interaction Component reads some value from the environment and sends the result to an Application Component. This component computes a control function and sends the result to a second Hardware Interaction Component that outputs the data. The single components interact only with the Broker. The task of the Broker is to forward the event to the relevant components.

[2]http://www.openarchitectureware.org/

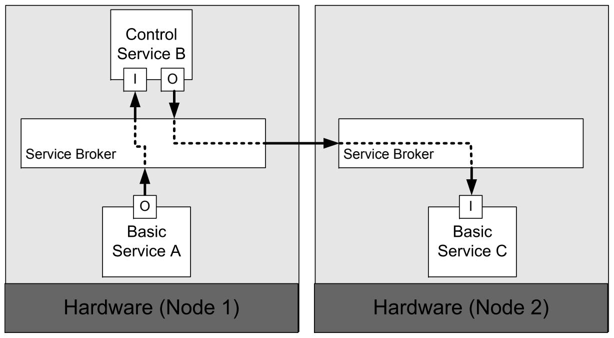

Figure 6. Example Application

We have implemented components to generate this middleware for the versions 1.1 and 2.0 of TinyOS. In addition, we also implemented components for Windows hosts that allow the easy implementation of graphical user interfaces to allow the interaction of the user with the sensor network. We use ZigBee for node-to-node, RS232 for node-to-host and UDP/IP for host-to-host communication. The physical connection is abstracted by the Network Service.

6 Application Example and Evaluation

The approach and developed tools were evaluated in the context of an example application realizing the control of a model railway, see Figure 7. For this application, we use MICAz [5] sensor nodes from Crossbow. Several Hardware Interaction Components were implemented to access the different available sensors: brightness, temperature, humidity and volume sensors. In addition, we also implemented a Hardware Interaction Component to enable the easy use of the MDA300 [4] data acquisition board of Crossbow that includes ADC and digital in- and output. These components are of course independent of the concrete application, we had in mind. They can be used in completely different scenarios.
In addition, we implemented different application components to calculate the speed and acceleration of the trains. As input, we used the data from the ADC and digital IO channels of the data acquisition board connected to different hardware devices (hall sensor, acceleration sensor). These components were implemented independent of the used platform.
Using these different components, we could implement the complete application. For example, we monitored the brightness to control the light of the trains for driving in the tunnels and during night. We also allowed the measurement of the train velocity. To demonstrate the interaction between the user and the sensor network, we implemented a signal-horn application. The user can use a graphical user interface running on a Windows PC to control the horn of the trains. The components realizing the interaction with the sensor network were similarly generated by our tool.

6.1 Evaluation

Several criteria can be used to evaluate our approach. We chose to compare our approach with a standard development process regarding the development time, the flexibility, the code size and the code maintainability. Two teams developed the same application. The first team implemented the application from scratch, while the second team used our code generator, but had to implement all the components (Hardware Interaction Components and Application Components) and templates for the middleware by themselves.
Both teams needed approximately the same time to develop the application. Not surprisingly, the first team could implement the first prototype earlier, since the second team had to implement the middleware and templates first. However, this initial effort was compensated during the development cycle. One reason for this surprising result was that by defining a meta-model and a middleware architecture, the implementation of the templates and components was straightforward. Of course, the advantages of our approach will be much more significant, when the templates and components are reused in further development processes. Regarding the code size, we expected some overhead of the generated code due to the middleware approach. Nevertheless the code size of both systems was of comparable size. The code developed manually had 310 loc and used 12 kB of flash memory; the generated code had 400 loc and used 13 kB of flash memory. The reason was that some functionality was repeatedly implemented in the different modules within the first solution. Due to the strict separation of concern, this problem was avoided in the generated code. Also the maintainability of the generated code was much better. Due to better design and documentation including the models, the readability of the code was significantly improved.
Regarding the flexibility, we could experience the forecasted flexibility. To support the MDA300 evaluation board, we had to switch to version 1.1 of TinyOS due to the unavailability of suited hardware drivers. As consequence, great parts of the code of the first development process had to be reimplemented due to the mixture of application and system logic. In contrast, in the model-driven approach only the templates had to be adapted to the new operating system, while the application components could be used unchanged.
Summarily, we could show that our approach has significant advantages. Especially when using the approach in the development of several applications, the development times can be reduced due to template reuse.

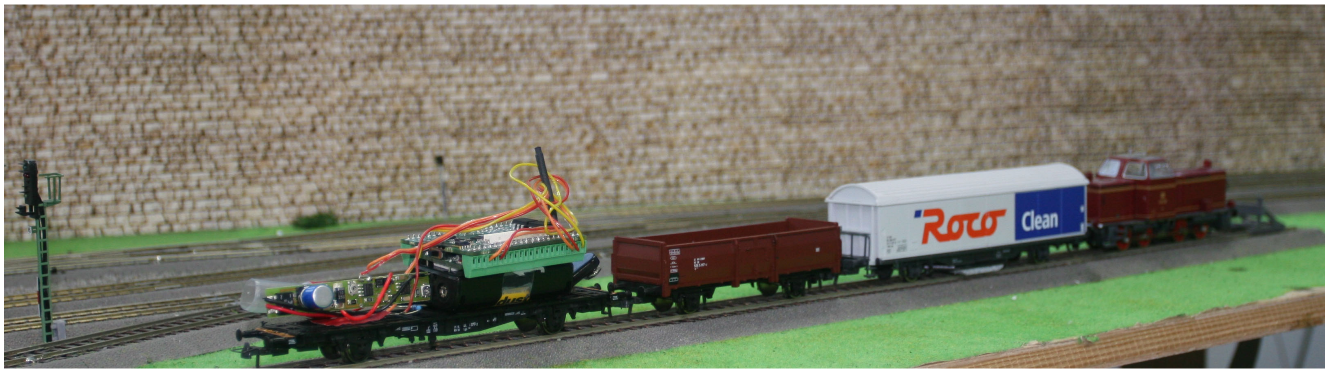

Figure 7. Application Example: Model Railway

	Standard Process	Suggested Approach
Development Times	o	+
Code Size	o	o
Flexibility	-	+
Maintainability	-	+

Figure 8. Evaluation Results

7 Related Work

Different research teams addressed recently the discussed issue by using macro-programming languages, middleware and component-based approaches for sensor networks [6, 15].
CORBA [14] is a widely used middleware standard, but the implementations are typically too resource consuming to be used in the context of sensor networks. The standards Minimum CORBA [13] and Real-Time CORBA [12] define a smaller subset to minimize these constraints. Nevertheless with a footprint of about 100 kB, the use of CORBA is not feasible for wireless sensor nodes. The .net MicroFramework [17] is with a footprint of about 300 kB in the same order of magnitude.
The OASiS Framework[11] aims at developing a framework that allows designing service-oriented sensor network applications. The design of applications is driven by an object-centric view, i.e., applications are designed in relation to a monitored object. This eases the development of monitoring or tracking applications, which require services to "follow" an observed object through the network. In contrast to our approach, OASiS does not provide automatic code generation.
The RUNES[2] middleware provides a component oriented programming platform for sensor network applications. The encapsulation in components with well defined interfaces allows dynamically reconfigure applications based on environmental changes, thus providing context aware adaptations. However, the design and composition of the individual components is still the task of an expert and cannot be done by the end-user himself. In our approach, the adaptation of the individual components is automated by the code generator.
Reusability is addressed in different standards for sensor networks. For home automation, the Konnex (KNX) [8] standard is used to ensure the interoperability of different devices. However, this standard specifies the hardware platforms that are allowed to use and is therefore not extensible. Furthermore, the standard does not address issues like automatic code generation or tool support during component development.
Industrial-process measurement and control systems can be implemented according to the IEC 61499 [7] standard. Different function blocks can be defined and a graphical user interface for application development is supported. Similar to our approach, the standard uses an event-based push model. However, the standard only addresses the system and application description and does not standardize the binary representation or the concrete application interfaces. Automatic code generation is not supported.

8 Conclusion

In this paper, we proposed an approach using domain specific languages and template-based code generators to generate an application specific middleware and to increase reusability.
For the domain specific language, we are using a component-based approach. The sensor network application is interpreted as a set of components that interact via an event based push model. Hardware Interaction Components

are used to access hardware devices and hide low-level implementation details. Application components implement the aspects of the application functionality and can be implemented platform independent. To form a concrete application, the interaction between Hardware Interaction Components and the Application Components must be specified in a model-based tool. The interoperability in heterogeneous systems with different nodes and also operating systems is realized by a tailored middleware.

The transformation of the model into executable code and the generation of the middleware are realized by our template-based code generator. Template-based code generators are designed to support an easy extension. Therefore, new templates can be easily added to support further platforms. In addition, the middleware can be augmented with new features using this extensibility.

The approach was tested in the context of a model railway. The implementation was done in two teams: one using standard methods, the other using the suggested approach. We could show the advantages of the domain-specific approach, especially regarding the flexibility in relation to the used hardware and operating system as well as the code maintainability. The development times were similar. The main reason here was the non-existence of the required templates for the middleware. We expect a significant acceleration of the development times, when using our approach in several projects due to reuse of templates and components. To prove this assumption, we will conduct a case study in the future. This case study will also focus on additional evaluation criteria such as power consumption and required execution times.

Since we just have started this work, there are a lot of features, we have in mind but could not yet implement. In addition to the Broker and Network Service component, we have to implement the other components mentioned in Section 2. In addition, dynamic reconfiguration can be used to cope with node failures at run-time. The sensor network can detect such failures and the reconfigure the network, e.g. by installing affected components on fault-free nodes.

Furthermore, we want to implement Quality of Service (QoS) methods into the network. Examples are the services that deliver the velocity and the acceleration of our model trains. To monitor the velocity, the user would typically select the sensor measuring the velocity. If this sensor fails, the sensor measuring the acceleration could be used as backup, but with a lower service quality due to measuring imprecision.

References

[1] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[2] P. Costa, G. Coulson, C. Mascolo, G. P. Piccoand, and S. Zachariadis. The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems. In *Proc. of the 16th Annual IEEE Intl. Symposium on Personal Indoor and Mobile Radio Communications (PIMRC'05)*, 2005.

[3] I. Crossbow Technology. Crossbow imote2.builder.

[4] I. Crossbow Technology. Mda300, data acquisition board.

[5] I. Crossbow Technology. Micaz, wireless measurement system.

[6] S. Hadim and N. Mohamed. Middleware: Middleware challenges and approaches for wireless sensor networks. *IEEE Distributed Systems Online*, 07(3), 2006.

[7] International Electrotechnical Commission. IEC 61499: Function blocks.

[8] International Organization for Standardization. ISO/IEC 14543-3: Information technology - Home Electronic Systems (HES) Architecture - Part 3: Communication Layers and Initiation.

[9] I. Johnson, C. Snook, A. Edmunds, and M. Butler. Rigorous development of reusable, domain-specific components, for complex applications. In *CSDUML'04 - 3rd International Workshop on Critical Systems Development with UML*, 2004.

[10] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[11] M. Kushwaha, I. Amundson, X. Koutsoukos, S. Neema, and J. Sztipanovits. OASiS: A Programming Framework for Service-Oriented Sensor Networks. In *International Conference on Communication System software and Middleware (COMSWARE 2006)*, 2007.

[12] Object Management Group. Real-time corba specification, Jan 2005.

[13] Object Management Group. Corba for embedded specification, version 1.0 beta 1 specification, Aug 2006.

[14] Object Management Group. Common object request broker architecture (corba) specification, version 3.1, Jan 2008.

[15] A. Rezgui and M. Eltoweissy. Service-oriented sensor-actuator networks: Promises, challenges, and the road ahead. *Comput. Commun.*, 30(13):2627–2648, 2007.

[16] SOS. https://projects.nesl.ucla.edu/public/sos-2x/doc/.

[17] D. Thompson and C. Miller. Introducing the .net micro framework, 2007.

[18] TinyOS. http://www.tinyos.net/.

[19] M. Torngren, D. Chen, and I. Crnkovic. Component-based vs. model-based development: A comparison in the context of vehicular embedded systems. In *EUROMICRO '05: Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 432–441, Washington, DC, USA, 2005. IEEE Computer Society.

[20] M. Voelter, C. Salzmann, and M. Kircher. *Model Driven Software Development in the Context of Embedded Component Infrastructures*, pages 143–163. 2005.

[21] M. Volter, A. Schmid, and E. Wolff. *Server Component Patterns: Component Infrastructures Illustrated with EJB*. John Wiley & Sons, Inc., New York, NY, USA, 2002.