ϵSOA – Service Oriented Architectures adapted for Embedded Networks

Andreas Scholz, Irina Gaponova, Stephan Sommer, Alfons Kemper, Alois Knoll
Technische Universität München
{scholza,gaponova,sommerst,kemper,knoll}@in.tum.de

Christian Buckl
fortiss GmbH
buckl@in.tum.de

Jörg Heuer, Anton Schmitt
Siemens AG, Corporate Technology,
Multimedia and Network Communication
{joerg.heuer,anton.schmitt}@siemens.com

***Abstract*—The development of embedded networks poses several challenges: complexity and size of the network, heterogeneity of nodes, and infrastructure dynamics. From other IT domains it is known that many of these challenges can be met by a Service Oriented Architecture (SOA). In the context of embedded networks, an application can be interpreted as a set of interacting services that produce, consume or manipulate data. However, the hard boundary conditions of embedded networks, such as limited resources and real time requirements, have to be taken into account. This paper discusses an embedded SOA (ϵSOA) concept based on the definition of an embedded service (ϵService) term and the differences to traditional Web services based SOAs. The paper describes a middleware platform that supports the execution and development of embedded network applications by employing model based code generation and a pattern based service composition model. The advantages of the approach are showcased using an application from the building automation sector, focusing on the energy management of smart buildings.**

I. Introduction

Embedded networks containing a multitude of networked nodes with varying sensing, acting, and processing capabilities are gaining increasing importance in many application areas such as the automotive, building management, or factory automation sector. Besides the challenges concerning the development of suitable hardware devices and communication infrastructures, the application development is increasingly difficult. The special characteristics of embedded networks, such as resource limitations, heterogeneous hardware, ranging from PCs over embedded controllers to primitive devices like switches, and the use of diverse communication protocols pose new and unique challenges. Analogous to other distributed systems, the development of customized solutions for every single installation is too costly and time consuming. A promising approach is to rely on the concepts of a Service Oriented Architecture (SOA): an application is interpreted as a set of data providing (sensors), data processing (application logic), and data consuming (actuators) services. Nowadays, Web services are the most prominent SOA implementation and have proven their suitability for building SOA based applications over the Internet. However, the notion of SOAs known from the Web service domain is not applicable for embedded networks, mainly due to hardware constraints.

The contributions of this paper are: (1) an analysis of embedded network requirements, (2) a rationale why traditional SOAs are not able to fulfill these requirements and the development of an adapted embedded SOA (ϵSOA) concept, (3) and the design of a corresponding middleware solution. These topics are covered in the following three sections. In Section 5, we present technical solutions for individual requirements and show how an integration between ϵSOAs from the embedded domain and SOAs from the Web service domain can be performed. In Section 6 we describe a prototypical realization of the ϵSOA approach for a building management scenario. In Section 7 we discuss related work. Finally in Section 8 we summarize the paper and outline open research topics.

II. Embedded Network Requirements

The term embedded network is used for a variety of setups which comprise diverse hard- and software components. In this paper, we will focus on embedded sensor-actor networks, which are used to perform control and automation tasks. Starting from these rough characteristics we identify the following detailed requirements:

Heterogeneity. A network built for automation purposes will typically contain nodes with a broad range of different capabilities. Depending on their task, nodes possess a diversity of processing, storage, sensing, and acting capabilities stemming from hardware components supplied by various manufacturers. Another source of diversity are user supplied devices which are used by the end-user to interact with the system, such as cell phones, PDAs, PCs, etc. This *heterogeneity* requires tools that allow building applications without prior knowledge of the exact hardware configuration, while simultaneously exploiting the given hardware resources as efficient as possible.

Distributed and Reconfigurable Architecture. In a control oriented network multiple distributed applications are simultaneously executed, each of them accessing a subset of the available sensors and actuators. As a consequence, a *decentralized network structure* is beneficial for control applications. It avoids the bottleneck of a single central node and ensures that not all applications cease to work if a single node fails. A distributed execution is also beneficial from an optimization point of view, because often the amount of transferred data can be reduced by placing the data consuming control logic nearby the data producing sensors. Furthermore, control networks have to be *reconfigurable at run-time*. At any time, new nodes with previously unknown functionality can be added. To support these dynamics, the network has to provide a repository of the available devices and a logging facility that allows to retrace changes. Because new applications can be installed at run-time, the purpose of individual nodes in the network is not fixed but changes throughout the lifetime of the network. This requires a dedicated *life cycle management* that supports the installation, startup, shutdown, and removal of applications on the nodes in the network.

Resource Limitations. Hard boundary conditions of sensor networks are *resource limitations* imposed by the underlying hardware. Consequently, an efficient execution of applications and compact network protocols are important. The diversity of the available hardware additionally requires *scalable functionality*. Small devices should only contain the bare minimum of functionality needed to perform their tasks, whereas more powerful nodes should be flexible enough to provide run-time adaptability.

Error Detection and Recovery. Node failure or communication problems are likely to occur in embedded networks, especially if battery powered devices or wireless links are used. Some problems can be compensated by the used network protocols, e.g., by re-routing data on alternative paths. Other exceptional situations, e.g., a non-functional sensor or actuator, may be compensable if redundant hardware is available. Development tools should support the creation of *robust applications*, which benefit from redundantly available hardware. Furthermore, foreseeable exceptional situations, e.g., energy depletion, should be detected and reported before an actual failure occurs.

End-User Programming. The applications running on a sensor network are typically not known in advance and often no trained personnel are available for the installation of new applications. E.g., an end-user, who wants to configure the mapping of lights and switches in his automated home, has neither programming experience, nor detailed know-ledge about the used hardware. Additionally, the applications executed on the network vary from installation to installation, because they depend heavily on user preferences and the available nodes. The opening of a broad mass market requires concepts which support an easy *end user programming*, i.e., enable an end-user to intuitively install, (re-) configure, and extend applications. Furthermore, *automation support* for the installation and configuration of applications in large scale installations is important. Subnets with similar functionality should only have to be configured once and similar installations should be configurable analogously to existing ones.

Bridging. Embedded networks do not operate in isolation but often possess access to wide area networks or the Internet. An easy integration of embedded networks with external components requires *Web service based interfaces*, as these are the de-facto standard for the communication with external services. The challenge thereby is to connect the Web service domain with its high resource demand and its highly available components to the embedded network domain with its small footprint nodes. Web service interfaces alone are not sufficient for the integration of embedded networks with enterprise back-ends, e.g., in a shop floor integration scenario. Additionally, the data delivered by the sensor network has to be integrated into the enterprise knowledge domain. This requires *semantic information* that allows to combine the measured data with the information contained in the back-end databases.

Summing up, we identified the following challenges for the development of embedded network applications:

- Heterogeneous hardware from various vendors
- Run-time adaptability
- Life cycle management
- Distributed execution of applications
- Resource efficient data processing
- Scalability to capabilities of the underlying hardware
- Error detection and recovery
- End-user programmability
- Automation support for installation and configuration
- Web service based interface for communication with external services
- Semantic support for enterprise integration

Besides these requirements, security aspects play an important role, especially if wireless communication is used. The implementation of security mechanisms is not in the focus of this paper. We believe that these are orthogonal to the presented solutions and have to be studied separately. In fact, many approaches known from the Web service domain might be transferable to embedded networks, given some adaptations to care for the limited resources on the devices.

III. ϵSOA

Service oriented architectures inherently support two major requirements mentioned in Section 2: heterogeneous infrastructures and run-time adaptability. Services provide a high level of abstraction that safely hides the implementation details of the hardware devices from the developer. Additionally, treating sensors and actuators as services allows dealing with the dynamics of the underlying network. Newly added devices provide services which can be automatically discovered and (semi-)automatically integrated into existing applications or used to build new applications. The benefits of SOAs known from traditional application fields such as enterprise service architectures can be translated to embedded network applications, too. The decomposition of applications into loosely coupled software modules provides high flexibility, re-usability, and extensibility and simultaneously eases the coexistence of different applications. Another benefit is the possibility to integrate services from various hard- and software vendors in a seamless way. Furthermore, due to the high abstraction level, domain knowledge is sufficient to intuitively understand the functionality of services and to install and (re-)configure applications in the network.

The special requirements of embedded networks discussed in Section 2 require some adaptations of the traditional SOA implementations like Web services:

Data Driven Services. While traditional SOAs are based on a request-response message pattern, control applications running on embedded networks are typically data driven:

data is acquired periodically at the sensors and pushed to connected services. These services produce new data based on the received input which is consecutively pushed to the next service in the processing chain.

Service Life Cycle. Service instances which abstract hardware devices, such as sensor services, may be used by multiple applications simultaneously. If one of these applications changes the state of the service, this is visible to other applications. Web service instances on the other hand are typically not shared and changes to one instance are not visible to other applications. SOAs for embedded networks therefore have to provide techniques to facilitate multi-user access.

Resource Contraints. Services used in the embedded domain will often be executed on small devices with severely limited storage and processing capabilitites, what requires efficient message formats for the communication between these devices. This can be achieved by a clear separation between the description of the data, which is specified only once, and the transmission format, which contains solely the raw data payload in a binary representation. Additionally services and the middleware they are executed in have to be desinged in an efficient way to facilitate the execution on small sized devices.

We will call services with these special characteristics embedded services (ϵServices), i.e., services implemented on small or embedded devices. The corresponding service oriented architecture consisting of these services is called ϵSOA. Embedded network applications based on an ϵSOA consist of a set of connected ϵServices. Each ϵService can either produce data (sensor ϵService), process data (logic ϵService), or consume data (actuator ϵService). The available hardware is abstracted as hardware services, which may be actuator ϵServices or sensor ϵServices. Logic services on the other hand do not depend on an underlying hardware and are executed on programmable control devices. The communication between services is based on data streams. A data stream consists of a sequence of data packets and connects an output port of an ϵService with an input port of another ϵService. In other words, ϵServices can be seen as operators on data streams, which produce data streams (sensor ϵServices), process data streams (logic ϵServices) or consume data streams (actuator ϵServices). This data flow model is translatable to an event based processing model on the nodes. The service logic is only executed upon the arrival of a new event, such as incoming data, an elapsed time period, or environmental events like a measurement value that exceeds a certain threshold. Between these invocations, the node can stay in a "sleep mode" that preserves energy resources.

IV. ϵSOA MIDDLEWARE DESIGN PRINCIPLES

Our ϵSOA middleware is based on a hierarchy of three views of the embedded network, as depicted in Figure 1.

A. Abstract Infrastructure View

The *Abstract Infrastructure View* provides a unified model of the available hardware and network connections. All hardware devices that are connected via a general-purpose network are represented by nodes. The nodes are connected by links,

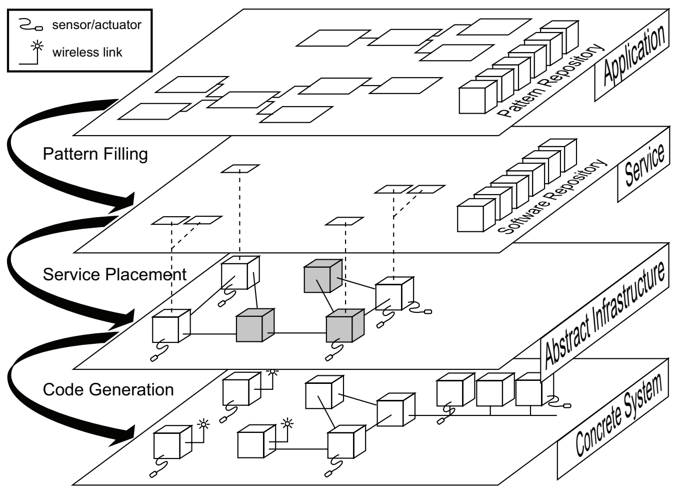


Fig. 1. Embedded Network Views

which represent a possible single-hop communication between two nodes. The actual transport medium and the communication characteristics are replaced by properties which are annotated to the links, e.g., bandwidth or packet loss rate. These properties and the links between individual nodes have to be provided by the underlying network protocols. Each node can possess several sensor and actuator devices: on-board sensors and actuators as well as devices that are directly connected to the node. Nodes without any sensing or acting capabilities may also be included in the network, e.g., gateways between different networks or programmable control units. The nodes are classified into two classes, programmable nodes which can execute arbitrary code (gray boxes) and nodes which possess only limited configuration options (white boxes).

B. Service View

The *Service View* provides a service oriented view of the available sensor and actuator hardware. Each sensing or acting device is represented by a corresponding hardware service. If a node possesses multiple sensors and/or actuators, this will result in multiple services. Additionally, the Service Layer contains a repository of available logic services. Each service is annotated with a set of metadata describing its characteristics: its general properties, such as measurement rates, its inputs and outputs, and the data that can be sent or received. This data is described by a data type and a reference to a domain specific taxonomy which defines the kind of data a sensor measures, e.g., "Temperature".

The metadata may have different originators. Hardware services will typically be described by the manufacturer, and logic services by the programmer of the service. During the physical installation of nodes, additional metadata can be entered by the installer, such as the position and orientation of sensors and actuators or management data like inventory numbers. The third category is dynamic metadata, which represents the current state of nodes, such as energy resources or utilization, and is monitored during run-time. Our ϵSOA middleware makes no assumptions about the presence of specific metadata fields, as these are heavily application specific. Instead it offers generic filtering algorithms that support the extraction, monitoring, and configuration of node subsets with given characteristics, e.g., all nodes which are in the same room or possess the same "floor" property.

C. Application View

The *Application View* supervises the applications running within the platform. It provides an overview of the installed applications, the contained service instances and the information exchanged between these instances. The Application View offers a repository with application patterns in order to ease the installation and configuration of applications. An application pattern describes an application as a composition of abstract services, called slots, and the data connections between these services. Every abstract service is defined by a metadata description, defining the required in- and outputs, their data types and characteristics. At run-time, the abstract services of a pattern are matched against the available hardware services and the logic services contained in the service repository in order to determine whether a specific application can be installed or not. The data connections between the available services can be fully automatically derived from the connections in the pattern. If the mapping of services to slots is unambiguous, i.e., there is exactly one service candidate for each slot, this mechanism allows to install applications fully automatically. If there is more than one candidate, the user has to manually select the appropriate one. If not all slots in the pattern can be filled the user can be provided with a "shopping list", which specifies the missing hardware components for the desired application. The user can compare this list with the available products of various vendors in order to determine which hardware possesses the required sensing or acting capabilities.

D. Application Development Workflow

The workflow for the installation of a new application is illustrated by the arrows on the left side of Figure 1. After the selection of a pattern and the assignment of suitable services to all slots, the middleware derives a "good" service placement, i.e., an optimized mapping of services to physical nodes. The optimization can be done for various goals, such as minimization of network load or homogeneous resource utilization on all nodes. The next step is the generation of platform specific code for the targeted nodes and the installation of this code on the nodes. During code generation, the specific characteristics of the node, e.g. CPU architecture and memory size, are taken into account to generate a compact and efficient service implementation. After that, the services are instantiated and configured according to the information provided by the application pattern, i.e., the destination for outgoing data is configured and service parameters such as thresholds are set.

V. Technical Realization

In this section we will outline some implementation details of our prototypical implementation of an ϵSOA middleware and provide technical solutions for the requirements presented in Section 2.

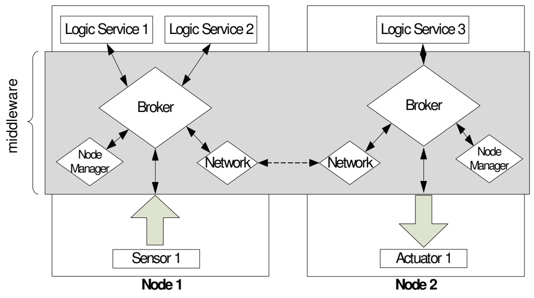


Fig. 2. Node Architecture

A. Efficient Distributed Data Processing

The constrained resources concerning processing power, memory, communication bandwidth, and energy resources require a middleware that supports an efficient collection, processing, and dissemination of data.

Generation of Efficient, Platform-Specific Code. Our platform deals with the heterogeneous hardware by using model based code generation, as presented in detail in [1]. The general node architecture is depicted in Figure 2. The main component is the *Broker*, which handles the data flow at the application level. The physical communication and data routing are abstracted by the *Network* component. The *Node Manager* implements the node announcement and service life cycle management. Services are embedded in the specific platform by the generation of a service stub. This approach facilitates a simple porting of services to other platforms as long as these provide a compiler for the same programming language. Another benefit of the code generation approach is the possibility to scale the middleware functionality down to the bare minimum required on a node, e.g., by using the techniques presented in [2]. Features not needed on every node are for example:

- the execution of dynamically loaded code
- support for specific network protocols
- monitoring or logging functionality

Event Based Data Processing. Sensor measurements are typically required in periodic intervals, e.g., once every 5 seconds. Between these measurements, energy saving techniques can be applied, e.g. turning off sensor hardware, putting the CPU to sleep mode, etc. Typical operating systems for embedded devices already support these features out-of-the-box. Our middleware caters for this requirement by providing an event based processing model. The execution of application logic can be triggered either periodically, by incoming data, or by environmental changes.

Distributed Execution of Applications. ϵSOAs inherently support the distributed execution of applications: all ϵServices which do not rely on specific hardware, i.e., typically all logic services, can be allocated at any programmable node. This allows optimizing the execution of applications for various criteria, such as resource consumption, reliability, responsiveness, etc. A big variety of data types can be used for the communication between ϵServices, ranging from simple measurements represented by a numerical value to complex business objects encountered during communication with the enterprise back-end. The middleware running on the nodes only provides data binding interfaces for the simple data types defined in XML-schema. If more complex types should be used, custom data binding code is generated and shipped with the service. This mechanism minimizes the overhead on small devices, but still offers the flexibility to use arbitrary data types on more powerful nodes. The external specification of the representation and meaning of data types by the metadata description allows reducing the transmitted data on the network to the raw data payload. This solution provides a transmission that is highly bandwidth efficient, however the transmitted data is not self-descriptive anymore. As a consequence, the compatibility between connected in- and outputs has to be checked during service composition, what can be done automatically based on the metadata descriptions. Of course it is still possible (but most likely computational too expensive for small devices) to dynamically infer the data type of an incoming data stream by inspecting the metadata description of the corresponding output.

B. Metadata Aided Service Composition

The metadata based description of services is beneficial for application development in two ways: It supports end user programming and provides automation support for service composition.

End User Programming. An intuitive end user programming interface is a decisive requirement for the acceptance of a sensor network middleware by non-expert users. A typical end-user will have little to no knowledge about the used middleware and the available software and often only very basic knowledge about the installed hardware. An example is a home owner who wants to build a light control application. This user will know that he needs some light switches, some light bulbs and some kind of control hardware, e.g., a programmable logic controller. While he will typically have no knowledge about implementation details, he will have a good understanding of the application domain, e.g., which switch has to be connected to which bulb. We think that application patterns are an intuitive programming paradigm for this kind of users. The user can browse a repository of pre-selected patterns, which are compatible with his hardware and choose a suitable one. Afterwards he can assign the available hardware services in a drag and drop manner to the slots defined by the pattern. The required logic services can be selected in a similar manner: the user simply chooses from a list of services with compatible metadata. The crucial observation is that both of the user's selections are solely based on domain knowledge represented in the metadata description. Nevertheless the created service composition will be guaranteed to work properly, as long as the metadata is correctly specified.

More experienced users may also develop own application patterns. This requires some basic understanding of the data types and properties of the used services, but no detailed knowledge about the implementation of the involved services. Patterns can be extracted out of existing service compositions or built from scratch by linking together services. A pattern should possess as few requirements as possible in order to keep it compatible with as many concrete services as possible, but still be restrictive enough to ensure the functionality of the composed application. Therefore the main task of the user is the selection of a minimal property set out of the properties present in the used services. Again, this decision is solely based on domain knowledge, i.e., it can be performed by a user with no programming skills at all. The newly created patterns can also be shared between end users. A community of users filling an Internet based repository of application patterns for various services is a promising vision for scenarios like building automation.

The development of logic services is well supported by the model based code generation: a programmer only has to fill a generated stub with application specific code. Simple services, e.g. a light control service, can be composed of basic services, such as comparators, adders, etc. and do not require manual programming at all. The user defined patterns may be shared with an interested community .

(Semi-) Automatic Service Composition. Automation support is important in application fields with lots of subnets of similar structure, e.g., rooms with similar equipment in a large building. The (re-)configuration of every subnet can be a tedious task and should be automated as far as possible. Application patterns provide a good starting point for optimizations. Changes to a single installation can be easily propagated to other installations that base on the same pattern. Furthermore, patterns can be inferred by inspecting the services in a given installation. Based on these patterns, suggestions for new installations can be made, providing a "copy & paste" like functionality. Depending on the amount of available metadata, this process can be performed fully automatically or interactively by asking the user to solve possible ambiguities.

C. Run-time Adaptability

Often embedded networks are not static but change over time when new nodes are added and existing nodes are removed or become unavailable. Another source of changes are mobile nodes, which enter and leave the network. Our ϵSOA platform supports these dynamics by providing a discovery mechanism for new nodes and the different system views presented in the previous section. Upon detection of a new node, the (semi-)automatic service composition can be used to provide a fast integration of the new hardware with the running applications.

The failure of nodes can be handled locally or globally. Local recovery is applicable if redundant ϵServices, i.e., ϵServices with a compatible metadata description, can replace a failed ϵService, or if redundant communication channels can be used by the network protocols. If this is not the case, a global recovery component can (semi-)automatically reconfigure the network, e.g., by switching to an application that does not require the failed node. This mechanism provides a graceful degradation to applications with reduced functionality instead of leaving the user with a non-functional installation.

The adaptation of nodes to new applications is only possible, if new ϵServices can be added to nodes at run-time. Our platform provides a management service which allows the (de-)installation, instantiation, startup, shutdown, and reconfiguration of services. Depending on the underlying operating system, the execution of dynamically loaded code may be possible or not. In the latter case, only services pre-installed on a node can be instantiated and started. In the former case, the compiled service is transmitted over the network and loaded by the operating system. In both cases, the newly created service instance is started and configured according to the application's requirements. If an application is removed, all services exclusively used by this application are shut down and de-installed.

D. Integration with External Services

The upcoming challenge for application developers is the integration of two worlds, Web services on the one side and embedded Services on the other side. A quick and seamless information flow between both domains ("real-time awareness") is growing in importance for manufacturing and logistics: a break in the information exchange between the embedded world and the business back-end is not tolerable anymore. Failures and delays on the device level have to be reported fast, in order to allow the timely execution of compensatory actions. Another example are highly flexible production environments, which have to be (re-)configurable from back-end services to reduce downtimes and support on-demand production.

The integration has to be performed in two ways: A developer familiar with Web service technologies should be able to interact with services from the embedded world just like he would interact with any other Web service. If a business process is modeled with BPEL[3], the process designer should be able to use ϵServices in the same manner as Web services in order to acquire or submit information to field level devices. On the other hand, a developer familiar with application development for embedded networks should have access to services in the enterprise back-end in the same manner as he accesses other embedded services. E.g., if data has to be transmitted to a back-end Web service, it should be sufficient to route the corresponding data stream to the remote service.

This mediation can be performed by a bridge service that converts incoming and outgoing messages and provides WSDL interfaces for the services contained in the embedded network. A detailed description of a possible implementation of such a bridge can be found in [4].

VI. Demonstrator

A comfortable home with an excellent quality of life is one of the main drivers in building automation and characterized by buzzwords like Intelligent House or Smart Home. Although the necessary technologies for realizing this vision are already available, general rollout has not taken place yet. One cause may be the significant installation costs especially in private households. On the other hand, sophisticated building automation is increasingly installed in industrial and business buildings, especially to lower management costs. Traditional building automation approaches emulate classical wired switched circuits and become significantly complex with an increasing or time-varying number of sensors and actuators.

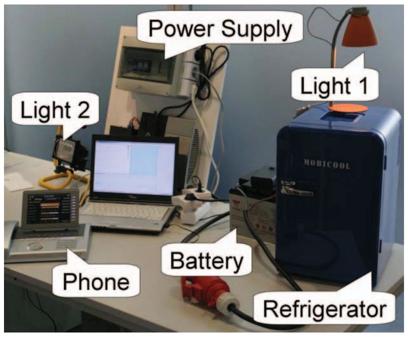
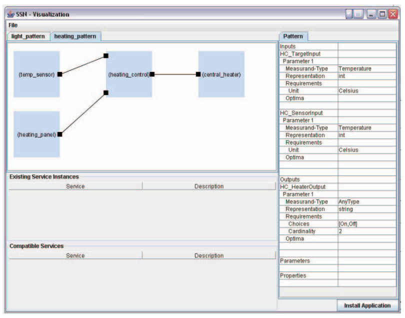


Fig. 3. Smart Home Demonstrator

Based on the ϵSOA platform described in the previous sections, we developed a demonstrator, which covers a future home automation scenario. The assembling of our demonstrator is shown in Figure 3. We assume that in the near future energy providers use dynamically changing energy prices in order to influence the overall energy consumption in a way that smoothes load peaks. We further assume that some kind of power storage system, such as the battery of an electric car, is present in future homes. We implemented the following scenario: A household comprising a battery and loads (a refrigerator and 2 lights) with different power consumption and energy saving options. One task of the automation logic is to minimze the energy costs throughout the day. If prices are cheap, the battery is charged and the refrigerator cools down to a lower threshold. If prices are high, the house is disconnected from the power grid and draws its energy from the battery. Additionally, the refrigerator is put to energy saving mode, i.e., it stops cooling until an upper temperature threshold is reached. There are other functional requirements not presented in detail here, e.g., the home has to connect to the power grid if the summed consumption of all devices exceeds the power of the battery, the battery should not be completely depleted, etc. The electricity prices are delivered by an external Web service, which is represented by a virtual sensor in the network. The used ZigBee based motes possess a set of I/O devices used to read signals from the switches and turn on or off the loads. The requirement to support end user programming can be also motivated by this example. Starting from a traditional control system, the user can add a battery and install a new pattern to benefit from the described price saving mechanism. Furthermore, the ϵSOA approach offers a new flexibility to end users. Changes, e.g. of the lighting, can be performed easily using the offered graphical user interface depicted in Figure 4. The demonstrator also points out how different devices can be used to administrate the embedded network. The programmable phone, for instance, can be used to monitor sensor readings and to adjust thresholds, such as the maximum temperature of the fridge. The experiences made during the development of the demonstrator revealed further optimization



Inputs	
HC_TargetInput	
Parameter 1	
Measurand-Type	Temperature
Representation	int
Requirements	
Unit	Celsius
Optima	
HC_SensorInput	
Parameter 1	
Measurand-Type	Temperature
Representation	int
Requirements	
Unit	Celsius
Optima	
Outputs	
HC_HeaterOutput	
Parameter 1	
Measurand-Type	AnyType
Representation	string
Requirements	
Choices	[On,Off]
Cardinality	2
Optima	
Parameters	
Properties	

Fig. 4. GUI for Pattern Installation

possibilities, which are presented in the last section.

VII. Related Work

There are standardized middleware architectures for specific application domains, e.g., KNX[5] for the building automation domain or AUTOSAR[6] for automotive applications. These approaches work on a very low abstraction level and therefore support neither end user programmability nor a seamless integration with external services, because the data processing paradigm is not compatible with service oriented principles.

There are middleware approaches for monitoring oriented sensor networks, e.g., TinyDB[7] or Cougar[8]. A typical characteristic of these systems is a hierarchical network structure, in which data is more and more aggregated towards the root. This infrastructure introduces unnecessary bottlenecks and single points of failure for control oriented applications involving multiple sensors and actuators.

A SOA approach for embedded networks is also persuaded by other projects, such as SIRENA[9] and SOCRADES[10]. These projects aim at making embedded devices directly accessible with Web Service technologies by installing an adopted Web Service stack, the DPWS[11] stack. While this approach is suitable for a certain range of devices, we believe that there will always be a class of very small and lighweight devices, which will not be able to deal with the additional overhead introduced by the Web Service technologies and therefore require a more efficient SOA implementation.

Other projects which apply a service oriented approach are OASiS[12], MORE[13], or RUNES[14]. We believe that our model based code generation and the use of application patterns allows better exploiting the characteristics of a given embedded network by generating tailored code and optimizing the placement of services.

VIII. Summary and Ongoing Work

In this paper, we presented the requirements and boundary conditions for the development of applications for embedded networks. We proposed an architecture based on an adapted service oriented architecture, the ϵSOA. We described the architecture of an ϵSOA middleware, outlined technical solutions for the individual requirements, and showed the feasibility of the approach for a prototypical implementation in a building management scenario.

Based on the presented principles further optimization is subject of ongoing research. The optimizations focus on resource efficiency of application execution, reliability of distributed applications and simplicity of application instantiation. The application execution can be improved by applying technologies known from data stream management systems, e.g., sharing of data streams between multiple applications. Different service placement strategies have to be evaluated and advanced heuristics for scenarios with little knowledge about the underlying network should be derived. Another direction for future work is the automatic learning of service patterns based on a repository of existing applications. Routing optimization with low overhead for protocols and routing tables can only be provided if application level connectivity is available at the routing layer. The development of such cross-layer network stacks is an interesting future research area. Furthermore we plan to enrich the semantic description of services to compute further metrics. These metrics may be used to select the most appropriate service out of a set of similar services.

References

[1] C. Buckl, S. Sommer, A. Scholz, A. Knoll, and A. Kemper, “Generating a Tailored Middleware for Wireless Sensor Network Applications,” *SUTC*, pp. 162–169, 2008.

[2] R. Pandey and J. Wu, “BOTS: A Constraint-based Component System for Synthesizing Scalable Software Systems,” in *LCTES*, 2006, pp. 189–198.

[3] “BPEL, Business Process Execution Language,” http://www-128.ibm.com/developerworks/library/specification/ws-bpel.

[4] C. Buckl, S. Sommer, A. Scholz, A. Knoll, A. Kemper, J. Heuer, and A. Schmitt, “Services to the field: An approach for resource constrained sensor/actor networks,” in *The Fourth Workshop on Service Oriented Architectures in Converging Networked Environments (SOCNE)*, 2009.

[5] KNX, “http://www.knx.org/.”

[6] AUTOSAR – Automotive Open System Architecture, “http://www.autosar.org/.”

[7] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, “TinyDB: An Acquisitional Query Processing System for Sensor Networks,” *TODS*, vol. 30, no. 1, pp. 122–173, 2005.

[8] Y. Yao and J. Gehrke, “The cougar approach to in-network query processing in sensor networks,” *SIGMOD Rec.*, vol. 31, no. 3, pp. 9–18, 2002.

[9] F. Jammes and H. Smit, “Service-oriented Paradigms in Industrial Automation,” in *IEEE Transactions on Industrial Informatics*, vol. 1, 2005, pp. 62–70.

[10] L. de Souza, P. Spiess, D. Guinard, M. Khler, S. Karnouskos, and D. Savio, “SOCRADES: A Web Service Based Shop Floor Integration Infrastructure,” *IOT’08*, pp. 50–67, 2008.

[11] Devices Profile for Web Services, “http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf.”

[12] M. Kushwaha, I. Amundson, X. Koutsoukos, S. Neema, and J. Sztipanovits, “OASiS: A Programming Framework for Service-Oriented Sensor Networks,” in *COMSWARE’06*, 2007.

[13] MORE – Network-centric Middleware for Group communication and Resource Sharing across Heterogeneous Embedded Systems, “http://www.ist-more.org/.”

[14] 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 *PIMRC’05*, 2005.