Modeling Distributed Signal Processing Applications

Werner Kurschl, Stefan Mitsch, Johannes Schoenboeck
Upper Austria University of Applied Sciences
Hagenberg Campus
Softwarepark 11, 4232 Hagenberg, Austria
{*kurschl,smitsch,jschoenb*}*@fh-hagenberg.at*

Abstract

Wireless Sensor Networks in general and Body Sensor Networks in particular enable sophisticated applications in pervasive healthcare, sports training and other domains, where interconnected nodes work together. Their main goal is to derive context from raw sensor data with feature extraction and classification algorithms. Body sensor networks not only comprise a single sensor type or family but demand different hardware platforms, e.g., sensors to measure acceleration or blood-pressure, or tiny mobile devices to communicate with the user. The problem arises how to efficiently deal with these heterogeneous platforms and programming languages.

This paper presents a distributed signal processing framework based on TinyOS and nesC. The framework forms the basis for a Model-Driven Software Development approach. By raising the level of abstraction formal models hide implementation specifics of the framework in a Platform Specific Model. A Platform Independent Model further lifts modeling to functional and non-functional requirements independent from platforms. Thereby we promote cooperation between domain experts and software engineers and facilitate reusability of applications across different platforms.

1. Introduction

Wireless Sensor Networks (WSNs) are applied in many different fields, e.g., environmental monitoring, pervasive healthcare, or sports training. WSNs could furthermore be used in the context of body sensor networks consisting of multiple interconnected nodes on, near, or even within a human body, which together provide sensing, processing and communication capabilities [1]. In comparison to traditional WSNs body sensor networks (BSN) are even more restricted in terms of hardware: they need to be smaller, less obtrusive and consume less energy. Moreover BSNs often comprise heterogeneous sensor types (e.g., physiological, biokinetic and ambient sensors) that need to cooperate together to measure a certain phenomenon.

[1] states that software engineers and domain experts, who are responsible to specify requirements, should be able to work together. In our opinion appropriate tool support is needed that enables this cooperation and hides hardware specifics from the domain experts. However, research to date focuses rather on low-level implementation details (e.g., routing protocols or time synchronization), and how WSNs can be applied in specific use cases. Therefore software engineers are often forced to use specific hardware and programming languages. This leads to applications that can hardly be reused in different use cases; tools or frameworks are scarcely available that help software engineers to reduce development time and to support them in the development process. An example for such a tool is YETI [2], consisting of a TinyOS textual and graphical editor. Further tools are compared in the taxonomy presented in [3]: they are all useful during nesC programming, but lack abstraction from low-level details. [4], however, states that "Raising the level of abstraction for programmers will be key to the growth of wireless sensor networks. Currently programmers deal with too many low-level details regarding sensing and node-to-node communication.".

A framework abstracting from sensor specifics is TinyDB [5] which provides a query language for sensor networks similar to SQL. It simplifies querying sensor values and also abstracts from different sensor platforms by means of the query language. Furthermore aggregation functions like minimum or maximum of sensor readings can be applied; but these functions are too restricted to derive low-level context from sensor readings, which hinders its use in body sensor networks. We argue that a more flexible framework is needed to efficiently make use of the local processing power of motes. They are not only able to read sensor data but also to locally extract features and derive context with classifiers.

The paper describes a signal processing software framework prototype based on TinyOS and nesC to build context-aware applications. Cooperation between domain experts and software engineers, however, cannot be promoted with such a framework. More abstract concepts are needed, which hide hardware specifics and programming languages. We propose that models could enable this cooperation. These models should not only serve documentation purposes, but rather be integrated into the development process as first-class artifacts [6], which can then be systematically transformed to source code—the main idea of Model-Driven

Software Development (MDSD). As BSNs may comprise several different platforms, the model can be lifted to further abstract from platform specifics. We therefore propose to include different layers of models: Platform Independent Models (PIMs) and Platform Specific Models (PSMs). A PIM is used by a domain expert to model the functionality. This PIM can then be automatically transformed to one or more PSMs that are refined by a software engineer. The integration of a PIM further raises abstraction, promotes cooperation between domain experts and software engineers, and allows the application of an MDSD approach in the field of context-aware applications in general and WSNs and BSNs in particular.

2. Related Work

Model-Driven Engineering (MDE) is a development approach using abstract models to describe systems; these models are systematically transformed to concrete implementations [7]. Model-Driven Software Development (MDSD) applies MDE with a focus on building blocks for software development processes. [8] defines MDSD as a general term for techniques that generate executable software from formal models. Model-Driven Architecture (MDA)—a specification of the OMG—is an MDSD approach based on the Meta Object Facility (MOF) to ensure interoperability between models. It emphasizes the separation between Platform Independent Models (PIMs) and Platform Specific Models (PSMs) as stated in [9]. Gaps between different models are bridged with transformations. [10] differentiates between transformations generating lower-level models from higher-level models (e.g., a PIM to PSM transformation) and mapping models at the same level of abstraction (e.g., PIM to PIM transformation). A prominent transformation approach is the Atlas Transformation Language (ATL) described in [11], which supports both imperative and declarative transformation rules.

In [12] pattern recognition systems are structured into five steps: sensing, segmentation, feature extraction, classification, and post-processing. Sensing collects raw sensor data, segmentation filters raw data to remove noise and undesired measurings, and feature extraction then reduces the data volume and prepares data as feature vectors. Classifiers decide to which category (context) a given feature vector most probably belongs to. During post-processing we can use domain knowledge to enhance the classifier's output and evaluate the costs that are incurred if we follow its decision. For detailed information on different context-awareness frameworks and context-aware systems please refer to [13].

SPINE is a TinyOS-based software framework for BSN signal processing applications [14]. The framework comprises signal processing components for sensor nodes running the TinyOS environment and Java components to manage the sensor nodes from a central coordinator node (star topology). The framework succeeds in abstracting from low-level TinyOS programming, but we still find that it could abstract from platform specifics even more rigorously.

The Context Recognition Network (CRN) Toolbox is a C++ framework integrating hardware abstraction, filter algorithms, feature extraction components and classifiers in a configurable runtime to support rapid development of context recognition applications [15]. It is designed for deployment to embedded devices that support the POSIX runtime environment. Although a graphical editor is provided it is not based on a formal meta-model and thus the CRN Toolbox cannot fully benefit from an MDSD approach.

The discussed projects and frameworks do not specify their architectures in formal meta-models. Therefore, they lack interoperability, development support, and software engineering methodology. Moreover, the projects make use of configurable runtimes, which in our opinion lead to undesired performance and memory losses. As we base our modeling environment on formal meta-models we abstract application development from specific platforms and can thereby generate code exactly tailored to a use case's requirements and to different platforms (which could be easily extended with SPINE and the CRN Toolbox).

[16] describes a modeling tool for wireless sensor network applications that supports the software development process with different levels of abstraction: domain experts model WSN applications in a domain model. Implementation is then detailed in a PIM; components are further refined in a PSM, before code for a specific platform is generated. The modeling tool is, however, too focused on wireless sensor networks to be directly applicable to body sensor networks; it does, e.g., not support cross-platform processing between sensor nodes and coordinator. In fact, their domain model corresponds to the PSM in our approach.

3. A TinyOS Signal Processing Framework

This section describes our software framework facilitating creation of signal processing applications on the TinyOS-2.x platform. TinyOS [17] is a component-based operating system for wireless sensor networks abstracting hardware resources in components. Its applications are programmed using the nesC programming language [18] based on modules and configurations. Modules represent component implementations whereas configurations are used to wire modules and other configurations together to form an application.

The nesC component specification is an ideal foundation for an extensible signal processing framework, which supports the steps in the pattern recognition lifecycle described in [12]: sensing, segmentation, feature extraction, classification and post-processing. The framework's components match these steps: *Readers* are used for sensing, *FeatureExtractors* for segmentation and feature extraction,

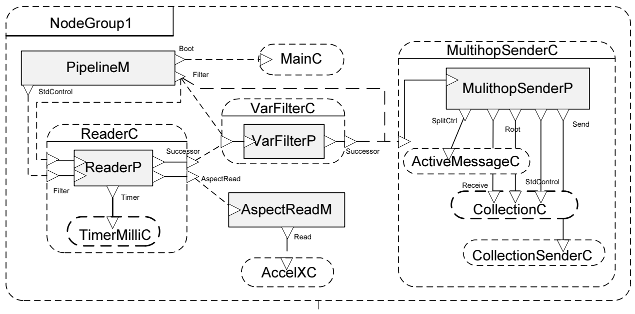


Figure 1. Sample signal processing application

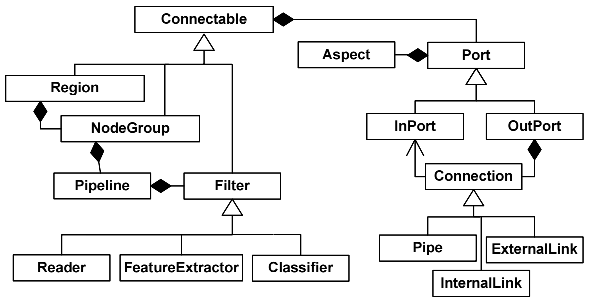


Figure 2. Platform specific meta-model

Classifiers for classification. Post-processing can be realized with feature extractors (e.g., smoothing the sequence of a classifier's predictions) or classifiers (e.g., multiple classification or voting). Wired together the components form signal processing pipelines according to the Pipes-and-Filters architectural style defined in [19].

The framework's foundation is the interface *Filter*. It specifies the *process* command to execute the filter's functionality such as e.g., reading data, extracting features, or classifying data. Every filter component in a pipeline uses and provides this interface. Thereby a preceding component can push data into its successive component. The wirings represent the pattern's *pipes*. New components can be integrated into the framework easily by implementing new modules, which have to provide and/or use the interface Filter. A configuration for the implemented module has to be provided to allow programmers to easily wire the new component together in an application. Fig. 1 shows the wiring of a sample application that reads horizontal (x-)acceleration values, applies a variance filter and broadcasts the result into the network using a multihop protocol.

Every pipeline starts with a reader delivering sensor data; we term semantically different data items *Aspects* (e.g., vertical or horizontal acceleration). Aspects are aggregated into *data packets*, pushed from component to component, and processed by every filter along the pipeline. The *AspectRead* interface wraps the TinyOS interface Read to assign a unique identifier to every sensor reading by which aspects can be distinguished. FeatureExtractor components modify data (e.g., a MinFilter, or VarianceFilter) and produce one output data packet for each incoming packet. Classifiers, in contrast, consume data packets and reason for low-level context, e.g., by applying a threshold. They produce new aspects for their successors.

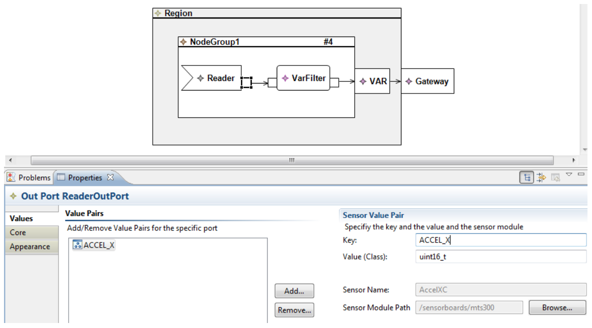
Besides filters that transform raw sensor data we provide utility components which are useful to form flexible applications. *Splitters* follow the Dispatcher Pattern in [20] to split up a data packet's aspects for different successors. The splitter's counterpart is the *Merger*: it aggregates aspects of multiple data packets. Communication between a WSN's motes as well as communication to a gateway is integrated into the framework by means of a *MultihopSender* component, which routes data packets through the network to a receiver. An *UARTSender* component transfers data to a PC.

The framework's filters are implemented in terms of modules and configurations. Therefore the main task of the software engineer is to wire components together to form a signal processing pipeline. Every component has to be wired to the *PipelineM* which implements the Boot's interface event boot to initialize all filters and to start the pipeline's reader. Furthermore the pipes between the components need to be established by setting the components successors (see Fig. 1). Nevertheless, programming nesC and other component-based platforms—especially defining component wiring—is error-prone and difficult to understand from source code, although tools try to simplify programming and provide editors that visualize the wiring. Unfortunately incorrect wiring might not always lead to compile errors but to semantic errors during execution which are even harder to detect as debugging is scarcely supported. Moreover such tools do not promote cooperation between domain experts and software engineers. As motivated in the introduction we propose formal models as abstractions from hardware and implementation details to enable Model-Driven Software Development (MDSD).

4. Platform Specific Modeling

A Domain Specific Langue (DSL), which focuses on the domain of distributed signal processing, simplifies using the described framework. In terms of MDSD DSLs are defined in meta-models specifying the abstract syntax of the domain. Fig. 2 shows the meta-model specific to the described signal processing framework (a PSM in the sense of MDA).

A TinyOS application comprises one or more pipelines. From a deployment view such an application can be hosted on several nodes that together form a *NodeGroup*. Node groups geographically close to each other are combined in a *Region*. Within a region communication between nodes is possible (the distance or differences in communication protocols between nodes in different regions prohibits direct communication) and represented by *InternalLinks*. Link de-



Out Port ReaderOutPort

Value Pairs
Add/Remove Value Pairs for the specific port
ACCEL_X

Sensor Value Pair
Specifiy the key and the value and the sensor module
Key: ACCEL_X
Value (Class): uint16_t
Sensor Name: AccelXC
Sensor Module Path /sensorboards/mts300

Figure 3. PSM model screenshot

tails (e.g., the routing protocol) are used to later generate the correct sending and receiving filter in nesC source code. If geographically separate regions want to communicate with each other, they are connected with *ExternalLinks*. The region's ports represent gateway applications translating from the in-network protocol to a different network, e.g., from Zigbee to a long range wireless protocol or to the serial port on a PC. Every *Connectable* instance (region, node group, and filter) defines interface contracts in the form of aspects on input and output ports (an aspect defines a data item's format and its semantics). Connections can only be specified between ports that share the same contract. The example wiring of Fig. 1 is now represented in the model shown in Fig. 3.

Models conforming to the meta-model represent signal processing applications on an abstract level. The graphical models are then used to create source code based on the described framework. The Reader in the model specifies the sampling interval; its out port specifies the aspect to acquire the horizontal acceleration and the nesC component actually handling sensor readings (cf. property view in Fig. 3). The reader in the model is transformed to ReaderC, AspectReadM and AxcelXC with the corresponding wirings. The TimerMilliC is derived from the sampling interval. The VarFilter is transformed to the equivalent VarFilterC. The pipe between Reader and VarFilter is transformed to a wiring from the ReaderC's used Successor interface to the VarFilterC's provided Filter interface. The NodeGroup1 aggregates the Reader and the Filter into a TinyOS application and its accompanying MakeFile. In the example the node group consists of four nodes as indicated by the number at the top-right of the figure. The nodes form a mesh-network which communicates its data to a gateway indicated with the out port VAR on the node group and the out port Gateway on the region. These ports are connected with an internal link; the link's routing protocol property is set to multihop communication. Therefore a MultihopSenderC instance is connected to the VarFilterC which is modeled by the pipe between VarFilter and NodeGroup1's out port. Every component is wired to the pipeline's used Filter

```
configuration NodeGroup1 { }
implementation {
    components PipelineM, MainC;
    components new ReaderC(1, 64) as Reader;
    components new AspectReadM(uint16_t,
       ACCEL_X) as XRead;
    components new AccelXC();
    components new VarFilterC(5) as VarFilter;
    components new MultihopSenderC(AM_VAR_ID,
       sizeof(DataPacket), GATEWAY_ID) as VAR;
    PipelineM.Boot → MainC.Boot;
    PipelineM.Filter → Reader;
    PipelineM.Filter → VAR;
    PipelineM.Filter → VarFilter;
    PipelineM.StdControl → Reader;
    Reader.AspectRead → XRead;
    XRead.Read → AccelXC;
    Reader.Successor → VarFilter;
    VarFilter.Successor → VAR;
}
```

Figure 4. Generated nesC wiring code

interface (thereby the pipeline can initialize every filter on startup). The ReaderC is furthermore wired to the pipeline's used StdControl interface to start data acquisition. Fig. 4 shows the source code generated from the model in Fig. 3 by applying the XPand template language [8].

Many applications, however, cannot be hosted on TinyOS nodes in isolation; instead, the TinyOS platform is only a part of a software system and cooperates with other platforms to achieve the application functionality (e.g., TinyOS sensors read data, preprocess it locally, and then transmit it via a gateway to a PC for presentation or further processing). Therefore, we raise the abstraction level to platform independent models described in the next section.

5. Platform Independent Models

The PSM described in the previous section still focuses on an implementation platform and therefore targets software engineers. By raising the abstraction level to platform independent models, we focus on the application's functional and non-functional requirements independent from hardware and software platforms, deployment details, and component framework implementations. Thereby, domain experts are able to express their application domain knowledge and the use case in a formal specification, instead of in informal requirements documents. The focus on requirements is twofold: Functional requirements are captured as signal processing components ordered in a pipeline. Non-functional requirements are described by properties on components.

Readers define the information available to the application in terms of output interface contracts. Their non-functional requirements capture user preferences and constraints as informal descriptions (e.g., a user may dislike image sensors due to privacy reasons but can imagine wearing accelerometer sensors possibly integrated into clothing). Additionally, the domain expert can observe characteristic behavioral patterns and environmental settings that indicate if a certain implementation is feasible or not. The functional requirements of feature extractors heavily depend on the use case. It is therefore impossible to represent every specific feature extractor within the platform independent meta-model. To gain modeling flexibility we provide two different variants: typed and generic feature extractors. Typed feature extractors are predefined in the platform independent meta-model; often, they are already implemented on many platforms. Rarely used feature extractors are integrated as generic feature extractors: they define their purpose, e.g., in terms of a mathematical function, and are implemented later on a specific platform. A classifier's functional requirements are defined with interface contracts, i.e., input aspects available to the classifier and output aspects it needs to derive. Additionally, the classifier type—either supervised, unsupervised or reinforcement learning algorithms [12]—and whether the classification problem consist of a process that unfolds in time suggesting e.g., a Hidden Markov Model, can be specified. Non-functional requirements describe the classifier's accuracy, precision, and recall.

The separation of the functionality of a system (represented in a PIM) and the adaption to specific platforms (represented in PSMs) requires automatic model transformations. WSN and BSN applications often comprise functionality that must be realized with cooperating platforms. For example, if we want to monitor an individual to detect behavioral patterns and present them to doctors, we would certainly use two platforms: TinyOS to read and preprocess sensor data, and a PC to host the user interface. Therefore, it is necessary to split a single PIM up into several PSMs during model transformation. The problem, however, arises which components of the PIM can be transformed to which specific platform. Additional problems are that multiple platforms could be candidates to host a component and that one platform could offer multiple specific types (e.g., the Java platform offers the writer types console writer and file writer). The first issue can be solved with formal platform descriptions, whereas the latter two need to be manually specified during transformation. This transformation specification is captured in a transformation model, which again conforms to a transformation meta-model. The transformation meta-model comprises the transformation specification of a PIM to one or more platforms. Each platform independent filter is transformed to a platform specific one (i.e., a PIM filter is a transformation's source model element, whereas the platform specific filter is a target model element). Fig. 5 shows the model transformation process in detail.

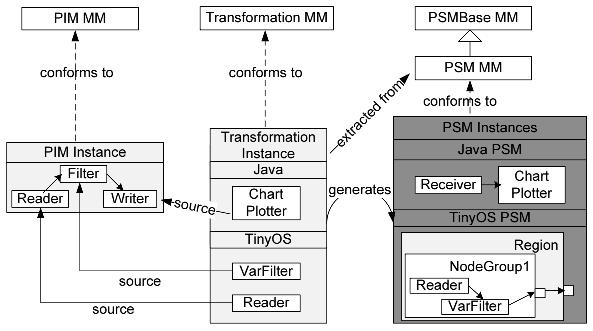


Figure 5. Transformation example

A PIM that conforms to the PIM meta-model captures the functionality (in this example, an aspect is read, filtered, and then presented to the user). At the beginning of the transformation process a transformation model instance is automatically created: it includes the available platform descriptions extracted from the PSM meta-models (the example shows an instance of the transformation model including the Java and the TinyOS platforms as possible transformation target platforms). The extracted platform descriptions are presented in a graphical wizard to guide the software engineer through the process of selecting possible platform specific realizations. The specified mapping is stored in the transformation model by adding source references (the PIM components) to target model elements, specifying the PSM model elements to be generated. After mappings for every PIM component are specified the transformation is executed to generate PSM model instances.

In case a PIM is transformed to more than one platform connections that bridge platform boundaries cannot be realized with in-process method calls (in contrast to connections within the same platform). Therefore we generate in the TinyOS PSM a Region with an output port representing a gateway (the connection's source platform) and a Receiver component on the connection's target platform respectively, as depicted in the generated PSMs in Fig. 5. Region out port and Receiver handle data marshalling and stream information between the two platforms. We used the Atlas Transformation Language (ATL, [11]) for specifying the transformations. For every supported platform a transformation from the platform independent meta-model to the platform specific meta-model has to be developed.

Domain experts do usually not have expert knowledge in programming, but should nevertheless be able to express their domain knowledge in platform independent models. Thus, an expressive mapping from the abstract syntax defined by the meta-model to specific machine representations and encodings must be defined: the so called concrete syntax. We used the Graphical Modeling Framework (GMF), see [21], to define graphical editors for PIMs and PSMs (cf. the prototype screenshot shown in Fig. 3).

6. Conclusion and Further Work

In this paper we presented a modeling approach for distributed signal processing applications, which are hosted by WSNs. The intrinsic complexity of such systems (different hardware platforms and programming languages, recognition of low-level context, and deployment options) requires abstractions in the form of component frameworks and models. In comparison to other modeling tools we additionally facilitate modeling across platforms with our notion of the PIM. Thereby, domain experts are able to express their knowledge on the application domain and use case independently from the implementation platform in a formal specification. By applying model transformations a PIM can be transformed to multiple cooperating PSMs which can be further refined by software engineers. This approach promotes cooperation between domain experts and software engineers.

The signal processing framework to date provides infrastructure to easily build TinyOS applications but provides only basic feature extractors and classifiers. Based on the implementation of use cases we will continuously evolve our framework. Experiences with the described modeling tools showed that there exists a wide conceptual gap between PIM and PSM which needs to be resolved in our ongoing work.

Acknowledgment

This research was supported by the Upper Austrian state government. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily represent the views of the research sponsors.

References

[1] M. A. Hanson, H. C. Powell, A. T. Barth, K. Ringgenberg, B. H. Calhoun, J. H. Aylor, and J. Lach, "Body area sensor networks: Challenges and opportunities," *IEEE Computer*, vol. 42, no. 1, pp. 58–65, 2009.

[2] N. Burri, R. Schuler, and R. Wattenhofer, "YETI: A TinyOS plug-in for Eclipse," in *Proceedings of the ACM Workshop on Real-World Wireless Sensor Networks (REALWSN)*. Uppsala, Sweden: ACM, June 2006.

[3] G. Teng, K. Zheng, and W. Dong, "A survey of available tools for developing wireless sensor networks," in *Proceedings of the Third International Conference on Systems and Networks Communications*. Washington DC, USA: IEEE Computer Society, 2008, pp. 139–144.

[4] J. A. Stankovic, I. Lee, A. Mok, and R. Rajkumar, "Opportunities and obligations for physical computing systems," *IEEE Computer*, vol. 38, no. 11, pp. 23–31, 2005.

[5] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: An acquisitional query processing system for sensor networks," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 1, pp. 122–173, 2005.

[6] J. Bézivin, "On the unification power of models," *Journal on Software and Systems Modeling*, vol. 4, no. 2, 2005.

[7] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *Proccedings of 29th International Conference on Software Engineering*. IEEE Computer Society, 2007.

[8] T. Stahl and M. Voelter, *Modellgetriebene Softwareentwicklung*. dpunkt.verlag, 2007, vol. 2.

[9] J. Miller and J. Mukerji, "MDA guide," OMG, 2003, last accessed 2008-09-11. [Online]. Available: http://www.omg.org/docs/omg/03-06-01.pdf

[10] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–645, 2006.

[11] F. Jouault and I. Kurtev, "Transforming models with ATL," in *Proceedings of the Satellite Events at MoDELS 2005*, vol. 3844/2006. Springer Verlag, 2005, pp. 128–138.

[12] R. O. Duda, P. E. Hart, and D. Stork, *Pattern Classification*. John Wiley and Sons Inc., 2001.

[13] M. Baldauf, S. Dustdar, and F. Rosenberg, "A Survey on Context-Aware systems," *International Journal of Ad Hoc and Ubiquitous Computing*, 2004.

[14] R. Giannantonio, F. Bellifemine, and M. Sgroi, "SPINE (signal processing in node environment): A framework for healthcare monitoring applications based on body sensor networks," January 2008.

[15] D. Bannach, O. Amft, and P. Lukowicz, "Rapid prototyping of activity recognition applications," *IEEE Pervasive Computing*, vol. 7, no. 2, pp. 22–31, April–June 2008.

[16] F. Losilla, C. Vicente-Chicote, B. Álvarez, A. Iborra, and P. Sánchez, "Wireless sensor network application development: An architecture-centric mde approach," in *ECSA*, ser. LNCS, F. Oquendo, Ed., vol. 4758. Springer, 2007, pp. 179–194.

[17] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, MA, USA: ACM, 2000.

[18] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proceedings of the International Conference on Programming Language Design and Implementation*. San Diego, CA, USA: SIGPLAN, 2003.

[19] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-Oriented Software Architecture 4: A Pattern Language for Distributed Computing*. John Wiley and Sons Inc., 2007.

[20] D. Gay, P. Levis, and D. Culler, "Software design patterns for TinyOS," *ACM Transactions on Embedded Computing Systems*, vol. 6, no. 4, p. 39, September 2007.

[21] M. Voelter, A. Haase, S. Efftinge, and B. Kolb, "Graphical modeling framework," *iX*, vol. 12, p. 17, 2006.