A Model-driven Software Development Approach Using OMG DDS for Wireless Sensor Networks

Kai Beckmann and Marcus Thoss

RheinMain University of Applied Sciences, Distributed Systems Lab,
Kurt-Schumacher-Ring 18, D-65197 Wiesbaden, Germany
{Kai-Oliver.Beckmann, Marcus.Thoss}@hs-rm.de

Abstract. The development of embedded systems challenges software engineers with timely delivery of optimised code that is both safe and resource-aware. Within this context, we focus on distributed systems with small, specialised node hardware, specifically, wireless sensor network (WSN) systems. Model-driven software development (MDSD) promises to reduce errors and efforts needed for complex software projects by automated code generation from abstract software models. We present an approach for MDSD based on the data-centric OMG middleware standard DDS. In this paper, we argue that the combination of DDS features and MDSD can successfully be applied to WSN systems, and we present the design of an appropriate approach, describing an architecture, meta-models and the design workflow. Finally, we present a prototypical implementation of our approach using a WSN-enabled DDS implementation and a set of modelling and transformation tools from the Eclipse Modeling Framework.

1 Introduction

Within the community developing complex distributed systems, the software crisis has been present for a long time already. For small-scale embedded systems, the situation is exacerbated by the fact that resources are scarce and thus code quality is rather defined in terms of compactness and optimisation for the specific platform than modularity and maintainability. This applies even more to software intended for cheap mass products with short life cycles.

As one possible escape from the crisis, model-driven software development (MDSD) has been proposed. MDSD promises code generation from abstract software models, which eliminates the error-prone manual design and implementation stages. As another important aspect for industrial mass production of embedded systems, MDSD can further help establishing product lines by managing line differences at an abstract, requirements-driven modelling level. In this paper, we will not further introduce the basics of MDSD; a good coverage of the topic can be found in [**?**].

MDSD has already found widespread use in business software scenarios like SOAs running on systems with relatively homogeneous architectures and comparatively abundant resources. Compared to those architectures, wireless sensor network (WSN) applications pose the additional challenges of close coupling of the code to the hardware platform including its peripherals, and a distinct lack of resources. Yet the prospect of being able to leverage MDSD for WSN environments is tempting, and in this paper we present an MDSD approach specifically designed to address those challenges.

WSNs heavily depend on the communication semantics used like publish-subscribe or event-based mechanisms, and on technical parameters like local node resources, transmission range, bandwidth and energy budgets. Regarding the communication aspects, we sought a solution whose semantics fit the requirements of sensor networks, that is, efficient data distribution, scalability and low overhead. We opt for the use of a data-centric middleware platform because we consider this paradigm a satisfactory match for WSN requirements, as described e.g. in [**?**]. Within this paper though, we will not elaborate on the pros and cons of communication semantics, but we state that we require that a data-centric approach be used for the application architecture targeted by our MDSD approach.

Instead of designing another data-centric middleware platform for our needs, we chose to build upon an existing standard, the Data Distribution Service for Real-time Systems (DDS) maintained by the Object Management Group (OMG) [**?**]. Choosing DDS is not only motivated by its being a standard, though. Besides being defined around a data-centric publish-subscribe paradigm, it offers a rich object-based API providing a base platform for application development. Unfortunately, although DDS is published as an OMG standard, it is not widely known (yet). Therefore, and because we tightly interweave DDS semantics and the metamodels used in our MDSD approach, we dedicate a section of this paper to describing DDS and its applicability to wireless sensor network platforms.

When describing our approach, we will use the terms "platform independent" and "platform specific" and the abbreviations PIM and PSM for the respective models to describe dependencies on hardware, operating systems and communications facilities.

The following sections present a review of publications related to our work and of the OMG DDS standard, followed by the description of our MDSD approach and the prototype created. Finally, we summarize the achievements, problems and prospective extensions.

2 Related Work

There are several recent research approaches that deal with different parts of the software development problem for WSNs. Using a middleware layer is an approved way to simplify application development for a WSN [**?**] [**?**]. Along these lines, various projects with different approaches and focuses have been published: Mires [**?**] is a message-passing middleware and applies the publish-subscribe paradigm to routing of data from data sources to data sinks. The provided API does not enforce a specific data model, and the definition of syntax and semantics of the messages are the responsibility of the application. Mires uses TinyOS and the protocols provided by it. With TinyDDS [?], an approach for a DDS-based middleware for WSNs was presented. TinyDDS is heavily based on TinyOS and nesC, which is the only programming language supported. Consequently, TinyDDS adapts the DDS API, and to some extent the semantics as well, to the event-based TinyOS. Finally, in [?], a model-driven performance engineering framework is proposed, which relies on TinyDDS. Using a configuration for application, network and hardware related features, the resulting performance is estimated and can be optimised without the need for run-time observation or simulations.

Model-driven software development is a recent promising attempt to address the WSN software development problem. Most of the related work for WSNs concentrates on the modelling of applications with subsequent code generation. In [?], applications are designed with domain and platform specific models and a component-based architecture description. The models are transformed to a TinyOS- and nesC-based metamodel. This is also the targeted platform, and its functionality is used by the generated application. [?] proposes the usage of meta programming languages for the definition of domain specific languages (DSL) suitable for domain experts. Algorithms or applications expressed in the DSL are platform independent and can be used for simulations or translated to a specific platform. The ScatterClipse toolchain proposed in [?] puts the focus on a development and test cycle for WSN applications. It uses an extension of ScatterFactory [?], an infrastructure for developing and generating applications for the ScatterWeb [?] WSN hardware platform. Besides the applications, the test cases are modelled as well and the application, environments for deployment and test, and modules are generated. The selection of required functionality during the generation process is performed at library level. The combination of MDSD with an embedded middleware system is proposed in [?], where a component-based middleware is tailored according to a model expressed in a purpose-built DSL. The middleware thus described connects hardware or application containers within the WSN. Specifically, platform independence is achieved by modelling the container interfaces with the DSL.

The usage of tools and frameworks of the eclipse project has become common among MDSD projects within the context of WSNs. But, so far, few approaches have been proposed which are using MDSD to generate an application specific middleware for WSNs. For embedded systems, MDSD has been an accepted technology for some time already. An example for a generic approach is proposed in [?], where a component/container infrastructure and the underlying middleware are generated in a MDSD-based process. Several DSLs are used to model the different elements, like interfaces, components and the overall embedded system. The application logic must be inserted manually in the generated infrastructure using the modelled interfaces. Another commercial approach proposed in [?] uses the abstraction of process variables as a base layer for implementing platform independent embedded applications for the automation context. The actual, platform specific data sources and sinks for the process variables are configured separately. Glue code to connect the local hardware or remote devices over different field buses can be generated in the process. Both approaches are using a dedicated interface for the application to access the generated infrastructure.

3 OMG's Data Distribution Service

DDS is an open standard for data-centric publish-subscribe middleware platforms with real-time capabilities published by the OMG [?]. The OMG is known for technologies like CORBA [?] and UML, and OMG efforts generally aim at portable, vendor-neutral standards. For DDS, this ensures that the scope of its semantics is not limited like ad-hoc solutions of research groups and single vendors. It can rather be assumed that DDS will remain an important player in the data-centric middleware landscape for a while.

Like CORBA, DDS uses an object-based approach to model DDS system entities, the API and communication data. At the heart of DDS data modelling, *topics* are used to denote possible targets of publications and subscriptions, like shown in Fig. **??**. DDS topic types are themselves modelled using the OMG IDL type set, offering complex types like structures and arrays and simple type atoms like integers. For a given topic, data publications and subscriptions exchange data samples with corresponding types. At the topmost level, DDS defines *domains* for application-level grouping of data types and communication domains. Domains also model the data space and thus the naming and addressing of topics and subordinate elements within a domain.

Fig. 1. Essential DDS architecture

Both the data model and the application interface of DDS are platform independent. They are specified in OMG IDL, for which language bindings exist that describe platform specific mappings. With these mappings, topic data and API elements are mapped to types, variables and methods, expressed in a concrete implementation language. Thus, the use of DDS alone already offers a high level of abstraction for data modelling and use of the communications API by the application, both of which are major parts of the problem space.

The DDS standard also describes a rich set of QoS requirements and guarantees that can be claimed and offered by communication partners, respectively. Among the areas addressed by DDS QoS policies are reliability of communication, bandwidth consumption, and latencies.

For the distribution of topic data between publishers and subscribers, DDS assumes a datagram service offering message integrity, and routing and broadcast capabilities. There is a recommended wire protocol standard (RTPS) [?]

published along with the DDS standard proper, which is supported by many DDS implementations.

The rich API and the design of RTPS suggest that DDS was not originally designed for small embedded wireless sensor nodes and networks with small packet sizes. To overcome the latter problem of RTPS, we also designed and realised an alternative DDS wire protocol (SNPS) optimised for WSN communications technology like ZigBee [**?**]. Additionally, subsetting of the DDS type space and API was necessary to achieve application footprints matching WSN node resources. Since this paper focuses on the MDSD aspects, we will not elaborate on those efforts here, but in [**?**] we were able to show that, on a technical level, DDS can be used in WSN scenarios. The general usability of the DDS approach for WSNs will be considered in the next section.

4 Approach

An MDSD approach generates executable applications from formal software models. For this, appropriate models to express the application and system design must be defined. This is done in turn using metamodels that describe the semantics and a syntax to define possible model elements. When an abstract syntax is chosen for the metamodels, one of a set of domain specific languages (DSLs) can be used to actually formulate a concrete model adhering to the metamodel.

In our approach, we further differentiate between platform independent models (PIMs) describing entities of the problem space and platform specific models (PSMs) targeting the solution space. With appropriate mappings, the transition from the PIM to the PSM domain can be automated such that human design activities can be restricted to the PIM level.

The main flow of MDSD activities is directed from PIM to PSM levels. As a refinement, although the ultimate output of a MDSD system should be application code, intermediate levels can be introduced that are already more specific in terms of platform specialisation (e.g. OS API) but have not yet reached the final code generation level. To restrict the impact of modifications of single model elements such that only the resulting model elements and code fragments affected must be re-generated, iterations within the design flow must be supported as well.

To summarize, the main building blocks of our approach are:

- a set of metamodels and DSLs describing the input models for the engineering process
- a pool of libraries and code fragments that constitute the composition elements and
- a tool chain consisting of parser and generator tools that process model information to assemble code pool elements to a deployable application

Although MDSD generally means to generate application code from abstract software models, the differences between business software scenarios and the architectures of WSNs must also be taken into account. For WSNs, an application must be regarded as a node-spanning entity because no single sensor node can achieve the goal of the application alone. Still, technical limitations concerning communication capabilities and node resources considerably influence the design of both the overall WSN structure and the software running on a single node. It is therefore desirable to capture the resources and non-functional requirements at the model level with the same priority as the semantic goals of the application.

From an MDSD output standpoint, WSNs require many instances of sensor node software to be built and deployed in order to form the overall application as the result of an orchestration of the application fragments. It might be tempting to initially assume that the code generated for all nodes can be identical, but the assumption of a strictly homogeneous hardware set-up and assignment of responsibilities limits the applicability of the approach. We therefore strive to support sets of nodes with heterogeneous capabilities.

4.1 Metamodels and Layers

To efficiently and completely capture the structure of a target application, we found that a single metamodel is not sufficient. Instead, we propose to diversify the metamodels available to the developer according to the different facets of the problem space to be solved. With this basic attitude, we could identify the following aspects to be modelled:

– problem domain-oriented typing of data to be exchanged
– topics as addressable entities for publications and subscriptions
– participation of a sensor node as publisher and subscriber within a domain
– QoS requirements for data exchange and node capabilities and
– hardware and operating system level configuration of a sensor node

Based on these requirements, we decided on four classes of metamodels to be available in our current MDSD approach: *data types*, *data space*, *node structure*, and *node config*. For the modelling of DDS *data types*, the PIM used is naturally based on OMG IDL [**?**], enriched with a topic type. Domains and the arrangement of topics within are independently modelled in a *data space* model, which can also capture QoS requirements to be fulfilled by the user of a topic. The roles of nodes as publishers and subscribers to topics are captured in *node structure* models. Again, optional model elements can be used to express QoS requirements for the node entities modelled. Finally, *node config* models are used to describe capabilities and resource limits of the physical node hardware and the operating system possibly running on a node.

4.2 DDS Integration

We chose DDS as the underlying data-centric middleware system. Actually, the current design of our approach depends on a combination of features we could not find in any other middleware approach that is currently available. Besides being a likely choice for a data-centric middleware because it is an OMG-supported standard, DDS is specified in a platform independent manner, which facilitates its adoption at the PIM layer of a MDSD framework. Additionally, DDS supports QoS requirements that can be exploited to match resource limitations of sensor nodes and communication channels. The API is object-based, and for applications focusing on data exchange, most of the application code activities are covered by the DDS API. For code generation, this means that most of the code generated will consist of the implementation of the DDS API and its invocations.

The usage of DDS also enables a variation of the approach which skips the generation of application code at least initially. Instead, the generated DDS interfaces are used during run-time through dynamic invocation from possibly hand-crafted code.

4.3 Architecture

The architecture of the MDSD framework is shown in Fig. **??**. From top to bottom, the degree of platform dependency increases. Input is given at the model layer which shows the four categories of DSL model descriptions that must be specified to define the application topic types, the DDS configuration describing communication relationships, and the system configuration at node level, respectively. Further input data from the left provides system-wide constraints, resource characteristics and pools of software fragments and is normally left unchanged across applications.

Fig. 2. Overall MDSD architecture

Descending from top to bottom, four lanes can be identified showing the metamodel and DSL elements participating in the parsing of the four model input categories into the still platform independent internal object representation. These object trees are subsequently checked for mutual compliance with QoS constraints defined in the model inputs, like resource limits in the configuration definition of a node. While still at the PIM level, optimisations based on information about the dependencies among DDS components can take place in the next step. This allows for re-use of DDS components within a single node.

The step from PIM to PSM is a combination of the transformation of application specific parts to program code, and the selection of DDS and system-level code fragments needed to support the application functionality. The actual program code to be deployed on a sensor node ultimately consists of the DDS application that is composed in a linking step from those prefabricated code fragments and the generated code.

4.4 Workflow

From an application designer's point of view, interaction with the development workflow shown in Fig. **??** happens at the model creation and modification stage, which is iterated with parsing and sanity checking steps until consistency is reached within the PSM set. The subsequent optimisation and transformation steps are carried out automatically, eventually delivering deployable application code as output. Up to now, this is the end of our MDSD workflow chain, which leaves to the user the actual deployment, testing, and feedback of testing results into the design stages.

Fig. 3. MDSD workflow

Many software engineering approaches rely on a cyclic model incorporating recurring phases in the application life-cycle. Because re-iteration of engineering steps is usually required after changes of requirements or fixing of design flaws, our approach strongly supports recurrent execution of single transformation and generation stages. The separation of model domains adds to the flexibility when adopting model changes because only the paths in the transformation tree containing changed models need to be considered. For monolithic and, thus, more closely coupled models, the impact of change is generally greater.

5 Prototype

The concepts described so far have been implemented in a prototype. For this, the following activities were necessary:

- selection of a target hardware and DDS environment for testbed applications
- selection and adaption of a set of tools for parsing and transformation of the models and for code generation
- definition of actual metamodels defining the PIMs
- development of code fragments serving as input for the code generator
- development of selectable code fragments for the DDS and OS level code repositories

Our implementation currently supports two target environments. For both, the DDS system (sDDS) and protocol implementation (SNPS) were developed in our laboratory [**?**]. To achieve fast turnaround times and allow for easy debugging and visualisation, COTS PCs and an Ethernet/IP/UDP-based protocol variant are used. For testing on an embedded target, we support an IEEE 802.15.4/ZigBee System-on-a-Chip platform (Texas Instruments CC2430) [**?**], [**?**] for the WSN nodes and versions of sDDS and SNPS matching the resource and packet size limits of the node devices. Of course, intimate knowledge of the implementation of a DDS system and access to the source code greatly facilitates the successful preparation of code fragments for the generator and the code repositories.

For the toolset, we chose the former openArchitectureWare tools Xtext for DSL definition, Xtend for model transformations, and Xpand for code generation, which are now part of the Eclipse Modeling Project [**?**].

The metamodels created basically adhere to the concept of four metamodel classes for data types, data space, node structure, and node configuration, as presented in the previous section. Up to now, for simplicity, we have only defined a single DSL that captures the features of all PIM variants. Deriving different DSLs from the metamodels can lead to greater flexibility and clarity in more complex implementations of our method, but the separate editing and processing of the PIM is independent from the number of DSLs used.

As a starting point for the creation of code fragments served a generic prototype application for our sensor nodes and our sDDS implementation. One of the advantages of using DDS is the inherent object structure, which naturally permits the separation of code fragments into object implementations. From the prototypical implementation, the structure of a generic sDDS application was derived; it served as a template for the implementation of the initialisation code generator.

Finally, a full-circle run of the prototypical tools in the development cycle delivered sDDS node application code implementing basic DDS data exchange functionality for the PC and CC2430 platforms, respectively, and the node applications could be deployed successfully. It could thus be shown that the inherent flexibility of the model-driven approach can be combined with the delivery of usable, compact output code. Especially the availability of only 128 KiB program memory on the CC2430 platform is honoured, and it should be noted that about 80% of the available code space is consumed by TI's ZigBee stack implementation, which drastically constrains the application's memory footprint allowance. For the basic functionality targeted here, the combined sDDS and application code footprint amounts to 25 KiB.

6 Summary and Future Work

Starting with the objective to facilitate software development for WSN architectures, we have presented an MDSD approach that relies on a set of meta models for differentiated requirements, and a communications and run-time infrastructure based on DDS.

The diversity of the metamodels used for specific facets of the requirements domain aids the expressiveness of the models created. Separate transformation of the models at the PIM level further facilitates the design of parser and transformation engines because the models are more specific, and thus, leaner. An intermediate, common PIM representation finally permits optimisation within the PIM space before the PIM to PSM transformation is applied.

DDS, that was chosen as a basis for the communication semantics and target application architecture, offers both a PIM representation for the modelling level and PSM mapping. The use of DDS also reduces the need for an additional rich OS API, and with the OMG it is supported by a major player of the professional IT community.

Using DDS for WSNs is a novelty, and we could not find any other approach combining MDSD with DDS and WSNs, but we could show that it is not only applicable but even beneficial if the DDS implementation and the transport protocol have been designed appropriately. Our implementation successfully targets a small-scale sensor node platform without sacrificing essential properties of the DDS core standard. This is an important aspect, since the DDS API can cater for the data processing design of a WSN application beyond mere data transport.

The basic design of our approach is defined at an abstract level, and we consider it to be sufficiently complete in its current state to be fully implemented without major changes to the basic ideas. Our implementation has not yet reached that coverage, though. The potential of diversifying the DSLs at PIM level has not been exploited, optimisation within PIM space is still rudimentary, and the set of available code fragments for the generation stages is far from exhaustive.

Furthermore, integrated support for the development of additional, possibly complex application code is not yet addressed by our approach. Here again, the usage of the PIM-level DDS API could facilitate an extension of the MDSD design to model the application logic based on an additional metamodel and DSL, and to generate tightly integrated, application specific middleware code.

We expect our MDSD implementation to grow with the completeness and the usage of the sDDS and SNPS implementations for WSNs in ongoing projects within our research group. Since SNPS is also available for IP/UDP environments, we hope to foster some interest in the general application of MDSD to DDS-based systems beyond the WSN scope.

References

1. Al Saad, M., Fehr, E., Kamenzky, N., Schiller, J.: ScatterClipse: A Model-Driven Tool-Chain for Developing, Testing, and Prototyping Wireless Sensor Networks. In: International Symposium on Parallel and Distributed Processing with Applications, pp. 871–885. IEEE (2008)
2. Al Saad, M., Fehr, E., Kamenzky, N., Schiller, J.: ScatterFactory2 : An Architecture Centric Framework for New Generation ScatterWeb. In: 2nd International Conference on New Technologies, Mobility and Security, pp. 1–6. IEEE (2008)
3. Beckmann, K.: Konzeption einer leichtgewichtigen, datenzentrierten Middleware für Sensornetze und eine prototypische Realisierung für ZigBee. Master-thesis, RheinMain University of Applied Sciences (2010), `http://wwwvs.cs.hs-rm.de/downloads/extern/pubs/thesis/beckmann10.pdf`
4. Boonma, P., Suzuki, J.: Middleware Support for Pluggable Non-Functional Properties in Wireless Sensor Networks. In: IEEE Congress on Services - Part I, pp. 360–367. IEEE (2008)
5. Boonma, P., Suzuki, J.: Moppet: Moppet: A Model-Driven Performance Engineering Framework for Wireless Sensor Networks. The Computer Journal (2010)
6. Buckl, C., Sommer, S., Scholz, A., Knoll, A., Kemper, A.: Generating a Tailored Middleware for Wireless Sensor Network Applications. In: IEEE International Conference on Sensor Networks, Ubiquitous and Trustworthy Computing, pp. 162–169. IEEE (2008)
7. Eclipse Modeling Project, `http://www.eclipse.org/modeling/`
8. Karl, H., Willig, A.: Protocols and Architectures for Wireless Sensor Networks. Wiley (2007)

9. Losilla, F., Vicente-Chicote, C., Álvarez, B., Iborra, A., Sánchez, P.: Wireless Sensor Network Application Development : An Architecture-Centric MDE Approach. In: Oquendo, F. (ed.) ECSA. LNCS, vol. 4758, pp. 179–194. Springer, Heidelberg (2007)
10. Masri, W., Mammeri, Z.: Middleware for Wireless Sensor Networks: A Comparative Analysis. In: International Conference on Network and Parallel Computing Workshops, pp. 349–356. IEEE (2007)
11. Object Management Group: Data Distribution Service for Real-time Systems, Version 1.2 (2007)
12. Object Management Group: Common Object Request Broker Architecture (CORBA) Specification, Version 3.1 (2008)
13. Object Management Group: The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification, Version 2.1 (2009)
14. Römer, K., Kasten, O., Mattern, F.: Middleware challenges for wireless sensor networks. ACM SIGMOBILE Mobile Computing and Communications Review 6(4), 59–61 (2002)
15. Sadilek, D.A.: Prototyping and Simulating Domain-Specific Languages for Wireless Sensor Networks. In: ATEM 07: 4th International Workshop on Software Language Engineering (2007)
16. Schachner, R., Schuller, P.: Zusammenstecken per Mausklick, Sonderheft Automatisierung & Messtechnik. Markt&Technik pp. 27–30 (2009)
17. Schiller, J., Liers, A., Ritter, H., Winter, R., Voigt, T.: ScatterWeb - Low Power Sensor Nodes and Energy Aware Routing. In: Proceedings of the 38th Annual Hawaii International Conference on System Sciences (2005)
18. Souto, E., Guimarães, G., Vasconcelos, G., Vieira, M., Rosa, N., Ferraz, C., Kelner, J.: Mires: a publish/subscribe middleware for sensor networks. Personal and Ubiquitous Computing 10(1), 37–44 (2005)
19. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. Wiley (2006)
20. TI: CC2430 A True System-on-Chip solution for 2.4 GHz IEEE 802.15.4 / ZigBee (2008)
21. Voelter, M., Salzmann, C., Kircher, M.: Model Driven Software Development in the Context of Embedded Component Infrastructures. In: Atkinson, C., et al. (eds.) Component-Based Software Development for Embedded Systems. LNCS, vol. 3778, pp. 143–163. Springer, Heidelberg (2005)
22. ZigBee Alliance, `http://www.zigbee.org/`