Model-Driven Development of Embedded Systems

Tino Brade, Michael Schulze, Sebastian Zug, Jörg Kaiser

[1]Department for Distributed Systems
Universität Magdeburg

Universitätsplatz 2, 39106 Magdeburg, Germany

tino.brade@student.uni-magdeburg.de

{mschulze, zug, kaiser}@ivs.cs.uni-magdeburg.de

Abstract. *Distributed mechatronic systems integrate sensors, processing units, communication networks, and actuators. In order to achieve a rapid development process and an improved maintainability it is necessary to combine and replace such modular components in a flexible way. For seamless composability we developed communication middleware and a programming abstraction for distributed sensors and actuators. In this paper we describe a comprehensive development toolchain based on these abstractions. Sensors and actuators are specified by an extended electronic datasheet for smart embedded devices. Following this approach, the user defines the capabilities of a device on a high system level in a declarative way. From that description, the functionality is generated using domain-specific tools like Matlab/Simulink. Finally, we improved the back-end tools that provide the code for the target system. Thus this code is derived with minimal user-intervention.*

1. Introduction

The development of distributed mechatronic applications requires an interdisciplinary effort of electrical and mechanical engineering as well as computer science. Such applications integrate heterogeneous systems like different sensor types, computers/microcontrollers and multiple diverse communication networks. To simplify and support the development process [Schulze and Zug 2008] describe a modular system structure. This allows combining components of different development stages, e.g. components in a preliminary stage, simulated by Simulink blocks with already tested hardware components (hardware-/software-in-the-loop systems). The modularity also supports testing and improves maintainability of the final product.

Precondition for this functional composability is a common abstraction for the components. We distinguish between three main aspects. Firstly, we need a generic communication interface, which decouples the application development from underlying networks and their specific characteristics. The second aspect is concerned with the internal structure of a component to provide a high degree of freedom when combining sensors, actuators and network interfaces. Thirdly, we suggest a method to integrate components that are defined and configured outside the development process (e.g. legacy components). We strive for an easy adaptation and integration of such components. A mismatch of configurations should be avoided by respective compatibility checks [Kaiser et al. 2008].

General communication mechanisms and interfaces are important for distributed applications. Usually, a broad spectrum of networks and lower level protocols has to be integrated. At a certain communication level, however, all elements of the distributed application have to agree on a common structured communication object. This requires the encapsulation of the underlying heterogeneous network structure. Therefore, we developed our communication middleware FAMOUSO (Family of Adaptive Middleware for autonomOUs Sentient Objects [Herms et al. 2008, Schulze 2009]) that provides event-based communication over different network types according to the publish/subscribe paradigm (CAN [Robert Bosch GmbH 1991], 802.15.4 [ZigBee Alliance 2003], Ethernet communication, etc.). FAMOUSO allows communication between components specified in different programming languages (C/C++, Python, Java, .NET) or by domain-specific engineering tools (LabVIEW, Matlab/Simulink). In contrast to other communication middleware, FAMOUSO supports different qualities of communication and is particularly developed for resource constraint devices as 8-bit controllers often used in smart devices.

A additional abstraction level addresses the internal structure of a sensor/actuator node. It describes typical internal modules required in smart devices. These include modules for data acquisition, signal conditioning, filtering and fault detection. Such a programming abstraction forces the developer to structure application specific code into modular and replaceable subsystems that e.g. can be represented and specified as blocks in Matlab/Simulink [MathWorks 2010a]. In [Zug and Kaiser 2009] we examined typical faults in sensor applications and suggested an architecture capable to cope with such situations. These concepts are also used in our proposed development chain.

Finally, we provide an abstract description of hardware and software configuration sets for a network node. This substantially simplifies and accelerates the development process. A standardized description also supports the use of code generation tools. Hence, changing parameters in some system component or even the integration of completely new hardware may be performed automatically or with minimal intervention only. As a result, the descriptions of the hardware components are capable to detect compatibility problems and may avoid faulty combinations. This is described in [Kaiser et al. 2008].

The aspects addressed above, i.e. communication, internal structures, and component description, represent the structure and the interfaces of a sensor/actuator node but do not reflect the component's behavior so far. The functionality of a component, e.g. the signal processing and the algorithms for filtering, has to be specified for each type of sensor and actuator specifically. This is the realm of domain specific programming languages and tools. Therefore, we combined the proposed abstractions with the Mathworks Simulink toolchain. This offers a large library of packages for control design and signal and state processing. Additionally, it includes tools (e.g. Real-Time-Workshop) to generate code from such a set of blocks for a specific hardware target. Our work includes an enhancement of such back-end tools for the AVR micro-controller.

The paper is structured as follows: In Section 2 we introduce the framework and illustrate the main concepts. Subsequently we use an example scenario to present the system descriptions of an appropriate node in Section 3 and the behavior and code generation for this application Section 4. Section 5 illustrates the state of the art and lists a survey of related approaches. Section 6 summarizes the paper and specifies current and future work.

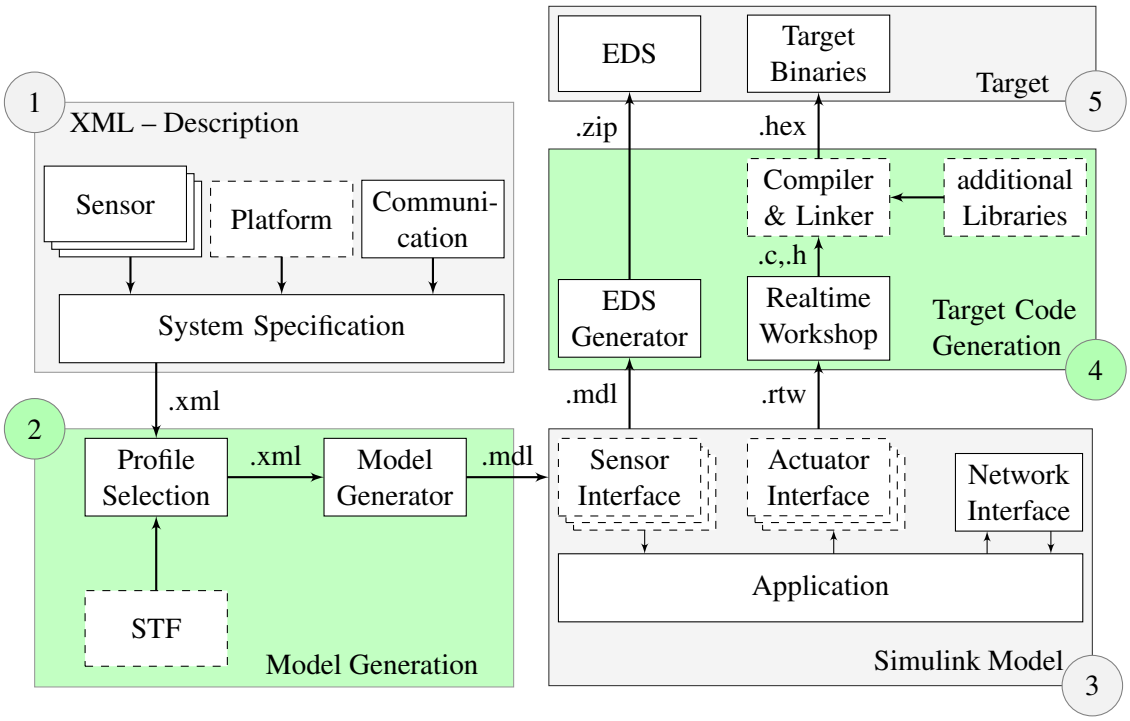


Figure 1. Framework development chain from XML descriptions to target code

2. Development Tool Chain for Distributed Applications

In this section we describe the main concepts of our development framework. Figure 1 illustrates the main workflow (1 - 5) and shows the three steps – *XML Description, Simulink Model* and *Target* – connected by two transformation/generation tools – *Model Generation* and *Target Code Generation*.

The box marked by (1) on the left site, *XML Description*, is the starting point of our development chain. Three different XML files include all information of the connected sensors. A platform specification defines the processor type, the board properties and the communication settings. Section 3 describes the structure and contents of those files. The clear separation of the descriptions in different files enables a flexible combination in varying scenarios. The *System Specification* determines the connection between sensor and board interfaces (an example is given in Section 3) .

Based on this abstract system specification we derive an appropriate Simulink model during the first generation step, marked by (2) in the workflow illustration. The *Profile Selection* module provides a flexible definition of the run-time environment of the Simulink model. This can be a simulation in the Simulink environment or an implementation for an embedded target. The *Profile Selection* algorithms check the availability on a Simulink *System Target File* (STF) necessary for the target code generation process and part of the Simulink model file. In a second step the users have to decide about the sensor inputs which can be implemented as simulation module or by an interface block to a real transducer. The sensor data sheets contain all information required for a basic sensor signal simulation. Hence, it is possible to coordinate simulated and real sensors on the target. The *Profile Selection* provides a graphical user interface and calls the *Model Generator*, that transforms the collected information into a Simulink file (.mdl).

We obtain the basic structure of a Simulink model ready for an integration of the *Application* methods as depicted in the right box labeled with (3). The *Application* processes input signals from simulated or real sensors as from the networks and calculates output values transmitted to *Actuator Interfaces*. At this stage the benefit of the framework is obvious. The developer does not need to cope with network or hardware interfaces. This is done automatically. It should be noted, that the user is able to control the *Profile Selection* from the *Model Generation* out of Simulink. The special block calls the generation process that derives a new Simulink model. Hence, simulated sensors can be replaced by real ones without any error-prone copy and paste actions between different model variants.

Simulink offers a comprehensive code generation tool chain, called *Realtime Workshop* [MathWorks 2010b], which was enhanced by a target configuration for a small 8-bit controller during a previous work [Brade 2009]. The necessary information for the target code generation process is stored in the Simulink model during the model generation process. Suitable compiler, linker and flash tools etc. are defined as well as additional libraries, run-time specification etc. so that it needs only a mouse click to bring the Simulink model down to the embedded target platform. Additionally, the Simulink model is used for the derivation of an Electronic Datasheet (*EDS*) of node's output done by our *EDS Generator*. It contains all information that is helpful for a correct interpretation, validation, and processing of the results transmitted by this node. A compressed version is stored on the target. This allows the use of service discovery methods for a dynamic integration and interaction.

In the following subsection we illustrate two parts of our framework more in detail, the XML description files for sensors and platforms and the profile switch mechanisms controlled out of the Simulink environment. For a comprehensible presentation we introduce an example scenario and assume, that we want to develop a smart temperature sensor. Our sensor node is equipped with an Atmel AT90CAN128 processor combines two types of temperature transducers, a AD592 and a CON-THEMOD with a higher precision but a smaller range. The AD592 offers only a voltage output while the CON-THEMOD module provides an additional digital I2C interface. The temperature signals should be jointed and the result published via FAMOUSO.

3. XML Description Files

The electronic data sheets were stored in an XML structure. XML offers a simple, standard way to exchange structured textual data. The advantage of this technique lies in the availability of machine processing using the Document Object Model (DOM) [Apparao et al. 1998] and the human readability in contrast to the binary representation of IEEE-1451. As depicted in Figure 1 we divide the entire sensor description into three types for sensor, platform and communication.

3.1. Sensor/Actuator Description

In a sensor data sheet we store general information of the transducer, interface description and context information. Due to the similarity, we handle both component types, sensors and actuators, with the same file type and structure. We are talking about sensors and sensor description in the following for the sake of simplicity and readability. However, similar statements are also applicable to actuators.

The first part of a sensor description file contains general information like sensor type, the vendor, layout pins and its supply properties similar to IEEE 1451.2. The interface description informs about all available interfaces and their configuration parameters that deliver sensor data or receive actuator commands. The third section is used to store context sensor properties. These properties describe signal behavior and parameters that are necessary for fusion and weighting mechanisms, simulation purposes and fault detection techniques concerning a reference output [Dietrich et al. 2010].

Considering our example scenario we have to write two sensor description files using an appropriate editor. This tool allows a convenient handling of the XML Files by a GUI that provides form structure with input boxes, buttons and drop down frames. The interface description list contains one entry for the AD592 sensor and two specifications for the CON-THEMOD, for each sensor statistical signal parameters and a linearization function are integrated.

3.2. Platform Description

The platform description combines basic information and the available interfaces of the platform. The first part has a similar background as the general information section from sensor description. Here we store general information like board type, board revision and processor type similar to CODES described in [Kaiser and Piontek 2006]. The second part encounters the interfaces and corresponding device drivers of the platform.

Our sensor board used in the example scenario provides the periphery of the Atmel processor and supplies interfaces to the analog digital conversion, I2C, and CAN buses, as well as two UART.

3.3. Communication Description

The communication description is tailored for the integration of FAMOUSO. It contains necessary information like subjects of the events for publishing and subscription as well as parameters like periodicity, omission degrees, etc.

In the example scenario the temperature values and their validity are published periodically.

3.4. System Specification

The three XML files mentioned above are composed in a system specification file. The specification file selects the interfaces between sensor components and the used target platform. Storing the connection data in a separate file results in a high degree of flexibility because it opens the ability to compose and replace different sensors, actuators and platforms. Consequently, it is possible to develop the application on different boards and select yet another one for series production. The system specification file is also generated by a graphical tool that helps to hide the error-prone task of manual XML editing and checks the compatibility of the interfaces. E.g. if a developer tries to combine a sensor with a LIN bus with our scenario board, this results in an error message.

Listing 1 summarizes parts of a system specification file. Line 4 and 5 define references to the connected sensors and their descriptions. As noted above, the first sensor (AD592) is assigned in line 10 to the third channel of the analog digital converter of our platform. The second transducer (CON-TEMOD-I2C) is connected to the I2C bus and uses the fifth address.

Listing 1. Example of a System Specification XML file

```
<?xml version="1.0" encoding="utf-8"?>
<container>
    <context>
        <Sensor1>AD592-BN</Sensor1>
        <Sensor2>CON_TEMOD_I2C</Sensor2>
    </context>
    <connections>
        <AVR-Processor>
            <Sensor1>
                <plattform>analog.channel3</plattform>
                <sensor>analog</sensor>
            </Sensor1>
            <Sensor2>
                <plattform>i2c.address5</plattform>
                <sensor>i2c</sensor>
            </Sensor2>
        </AVR-Processor>
    </connections>
</container>
```

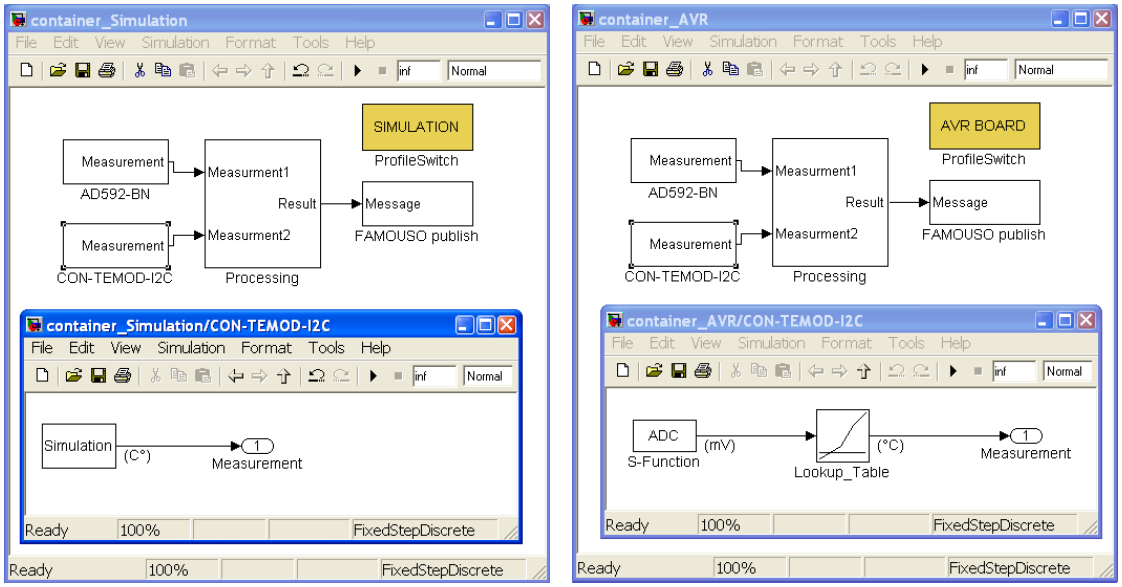
4. Simulink Model Generation

The model generator transforms the information of the specification file into a Simulink model. Beside the visible model structure of a node, the process generates the context information, device drivers for sensor components and the scripts necessary for profile switches. For this purpose a configuration set for target code generation is selected and the user can customize the sensor interfaces with the respective graphical tool. The configuration set, called profile, contains device drivers and the Simulink parameterization for the associated platform. In our example project we use two general profiles: the simulation in the Simulink environment and the execution of node behavior on an Atmel AVR processor. Within real target profiles the developer can replace physical sensors interfaces by simulation blocks.

A Simulink model combines different blocks connected by arrows that represents the data flow as shown in Figure 2. A block is presented by a rectangle and hides its functionality behind the respective graphical representation. The developer may describe each block by an individual S-Function, by a model reference or including a common library block.

Figure 2 shows the output of the generation process. We obtain a basic structure that offers suitable interfaces to the sensors, actuators and to FAMOUSO. The developer implements the behavior of the *Processing* block only and integrate the desired application here.

The Simulink models in Figure 2 of our example scenario look similar on the highest level, but for the different profiles two various sub-models implement the interfaces of the AD592 and CON-TEMOD-I2C sensor (visible in the sub window in each figure). In Figure 2(a) we depicted the simulation profile. The actual state of profile switch is



(a) Simulation Environment (b) Code Generation Configuration

Figure 2. Profile switch between a simulation environment (a), and a code generation toolchain (b)

documented in the headline of the *ProfileSwitch* block. For simulation practice, there is only an emulation driver with the output unit "Degrees Celsius". On the right side, we use the profile *AVR BOARD* connected to the real sensors. Hence, the first block named S-Function represents the analog device driver for the Atmel processor. The device driver delivers a signal with a voltage, which have to be transformed in "Degrees Celsius" and linearized by a "Lookup_Table" block.

After the implementation of the behavior and validation by simulation, the target code generation process translates a Simulink model into the target languages. The process of code generation is controlled by a System Target File (STF). One STF represents a range of processors, typically a processor family. The Embedded Real-time Workshop [MathWorks 2010c] provides a number of these STFs. However, in most case, especially for 8-bit devices, there are basic structures only. Hence, we created our own STF for the AVR that is now available for future projects [Brade 2009].

5. Related Work

Our approach covers a broad spectrum of methods used in system development and a number of ongoing research topics. Hence, it is difficult to compare our work to other projects under a single perspective. Therefore, we evaluated several tools, standards etc. that include ideas related to our approach. In Tab. 1 we summarize the results according to the three abstraction categories of our approach. An empty field in the table means that this feature is not supported, a "+" marks some basic support while a "++" denotes the comprehensive integration of an aspect. The "✓" symbol validates the existence of the main feature categories.

In the first group of categories we determine the integration of a communication abstraction that offers a common interface and helps to hide specific properties of the

		CODES	SensorML	OMG STIS	IEEE-1451.2	LabView	Simulink	CANopen
Communication Abstraction		✓		✓	✓			
Middleware Integration		++		++	+			
Modular Concept		✓			✓	✓	✓	✓
Module	sensor interface	+			++	++		
	communication	++				+	+	+
	functionality				+	+	++	
Electronic Data Sheets		✓	✓	✓	✓	✓		✓
EDS	sensor		++		++	++		+
	platform		++					+
	communication	++	+					+
Description Languarge		XML	XML	XML	binary	binary		text
Functionality			✓			✓	✓	
Programming Interfaces		C	Math ML, Java			C, g, m, etc.	m, mex, etc.	
Code Generation		✓				✓	✓	
Code Generation Tool		XSLT					RTW	
Target Languarge		C				C	C	
Error prevention		+				+		

Table 1. Comparison of different development environments for distributed applications considering high level sensor descriptions

underlying network. Some of the referenced approaches provide or integrate an existing communication middleware for this purpose. The second group categorizes the existence of a modular structure, which combines predefined functions like sensor and communication interfaces, error detection modules, as well as application specific modules. The next category examines the usage of an abstract component description in electronic data sheets for different categories of sensor node components. Category four divides the references considering the implementation interfaces for behavior related functionality. The utilization of this information is depicted in category five. Here we mark the capabilities of a code generation and combination in the development process. The following tools and approaches were classified according to the four categories:

CODES (COsmic embedded DEvice Specifications) described in [Kaiser and Piontek 2006] represents a predecessor of some parts of our framework. The approach focuses on a XML based description language for the specification of sensor features and communication parameters for smart autonomous components. The communication abstractions of the underlying middleware COSMIC [Kaiser et al. 2005] are mapped to the electronic data sheet and allow a dynamic setup of the communi-

cation. The sensor descriptions follow some ideas of the Transducer Electronic Data Sheets (TEDs) according to unit coding, data types, and boundaries. The CODES data sheet is compressed available on each node. CODES supports an underlying middleware extensively, but it does not consider realistic sensors and whose parameters in the data sheet design.

Sensor Model Language (SensorML) provides a framework for describing sensor systems, as well as the associated data processing. In contrast to the following standards the user can define filter or fusion functions in the Mathematical Markup Language (MathML). The comprehensive concept is used to describe sensor, platform, and functionality. The description of process properties is based on Sensor Web Enablement Common namespace and provides the entire Sensor Web Enablement functionality. The communication interface would be described by an extension of OSI. The concatenation of separate processes enables processing, analysis, and visual fusion of multiple sensors. SensorML does not include appropriate tools for behavior development beside the MathML interface. For complex fusion applications an abstract description of algorithms is not possible in this way.

The OMG Smart Transducer Interface Specification (STIS) [Object Managment Group (OMG) 2003] provides an access via the CORBA real-time service (RS) interface, the diagnostic and management (DM) interface, and the configuration and planning (CP) interface of small, smart transducers in a distributed control system. The standardization of the different interfaces is mapped on an interface file system (IFS) typically in the memory of each Smart Transducer. For an interpretation of the data in the IFS additional meta data about the particular IFS are stored on a central node with higher performance. The authors of [Elmenreich et al. 2004] enhance the Standard Transducer Interface (STI) concept and developed a XML description of the functionality for simple fusion tasks. As mentioned one section before, OMG STIS divides interface programming and application development. The descriptions are used for message identification but not in the development process.

IEEE 1451 Smart Transducer is a family of standards for connecting smart devices [IEEE Standards Association 1997]. IEEE 1451.2 defines an electronic data sheet and a digital sensor interface to access sensor measurements, set actuators, control maintenance functions, or to obtain the data sheet of the sensor/actuator system. Hence, the standard establishes the communication between a Network Capable Application Processor (NCAP) and an actual sensor node called Smart Transducer Interface Modules (STIM). Those structure represents a mechanism that enables a flexible network interface via special NCAPs. The standards 1451.3 to 1451.5 enhance the interaction between STIMs and NCAPs to various protocols and interfaces. The description of the sensors, stored at each node contains a detailed specification of the sensor's vendor, firmware, and physics in a compressed TEDs [Char 1997]. Tools for an additional use of the electronic data sheets beside message identification and interpretation are not known yet.

Mathworks Simulink [MathWorks 2010a] and National Instruments LabVIEW [National Instruments 2009] are widely used toolchains for simulation, Hardware-in-the-Loop (HiL) scenarios and code generation. Therefore a broad variety of tool-boxes (e.g. control engineering, data acquisition, image processing, etc.) are available and helpful for rapid developments. Both tools offer interfaces for different programming languages

beside the standard graphical oriented development systems. Simulink does not support the utilization of meta information about data sources like sensors. The code generation process checks only data types of the used variables. LabVIEW integrates the concept of TEDs from IEEE 1451 and identifies connected sensor based on this information. The user has the possibility to scan the network during the development process and to call calibration functions. Data sheets can be located on the node or in an extended version as a virtual TED on a server application. Simulink and LabVIEW offer a large amount of development tools in particular for code generation. However, they do not allow an external description of the used sensors, processors etc. All parameter have to be defined directly in the model.

CANopen [CAN in Automation 2005] is a high-level protocol for CAN-bus [Robert Bosch GmbH 1991]. Every CANopen device is delivered with a vendor electronic data sheet. This electronic data sheet specifies communication, error and application profiles. The developer can define profiles to customize a CANopen device. These modifications are deposited by a device configuration file. The exchange of information occurs by a process data object and a service data object. Process data objects carry the real-time data with a look-up mechanism to encode units. In contrast, service data objects were used to configure the CANopen device. All configurations of a device were placed in the device object dictionary. Thus, the device object dictionary is the abstraction between application and communication.

From Tab. 1 we can conclude that none of the related approaches meets all requirements completely. Each of the presented tools, standards etc., covers an individual focus only and shows excellence in just this point. While engineering tools like Simulink and LabVIEW do not consider (or in very limited extent only) external knowledge about sensors and communication specification, they are very suitable for developing the respective control and filter algorithms. Additionally, incoming data structures have to be correctly interpreted by the developer. The standards for smart transducer interfaces define the communication, services, data types, etc., but they do not care about the relation between input and output values. Hence, we have to combine several approaches to meet our requirements addressed in Section 1.

6. Conclusions and Outlook

The intention of our framework is to enable a flexible combination of different software and hardware components during a development process. Furthermore we aim at the integration of domain specific tools in distributed control applications composed from networks of smart sensors and actuators. Therefore, we introduce abstractions of processors, sensors and actuators defined in XML descriptions and connect them with widely used development tools. This enables the developer to configure sensor and network interfaces on a high, declarative level. As a result applications can be developed independently. The developer may also choose the favourite domain specific language.

In the future we will test and refine the framework in a distributed robotic scenario. Additionally, we want to develop an appropriate way to derive the *System Target File* automatically for code generation using a XML definition file too.

Acknowledgement

This work has partly been supported by the Ministry of Education and Science (BMBF) within the project "Virtual and Augmented Reality for Highly Safety and Reliable Embedded Systems" (VierForES).

References

Apparao, V., Byrne, S., Champion, M., and Isaacs, S. (1998). Document object model (DOM) technical reports : Level 1. Candidate recommendation, W3C.

Brade, T. (2009). Codegeneration aus Simulink / Embedded Real Time Workshop Modellen am Beispiel eines AVR Targets, Student Research Project, Otto-von-Guericke Universität Magdeburg.

CAN in Automation, editor (2005). *CiA 306 DS V1.3: Electronic Data Sheet Specification for CANopen*. CiA, CANopen.

Char (1997). *IEEE Std 1451.2-1997, IEEE Standard for a Smart Transducer Interface for Sensors and Actuators*.

Dietrich, A., Zug, S., and Kaiser, J. (2010). Detecting external measurement disturbances based on statistical analysis for smart sensors. In *Procedings of the IEEE International Symposium on Industrial Electronics (ISIE)*.

Elmenreich, W., Pitzek, S., and Schlager, M. (2004). Modeling distributed embedded applications on an interface file system. In *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 175–182.

Herms, A., Schulze, M., Kaiser, J., and Nett, E. (2008). Exploiting publish/subscribe communication in wireless mesh networks for industrial scenarios. In *Proceedings of Emerging Technologies in Factory Automation (ETFA '08)*, pages 648–655, Hamburg, Germany.

IEEE Standards Association (1997). *IEEE Standard for a Smart Transducer Interface for Sensors and Actuators (IEEE 1451.2)*.

Kaiser, J., Brudna, C., and Mitidieri, C. (2005). COSMIC: A real-time event-based middleware for the CAN-bus. *Journal of Systems and Software*, 77(1):27–36. Special issue: Parallel and distributed real-time systems.

Kaiser, J. and Piontek, H. (2006). CODES: Supporting the development process in a publish/subscribe system. In *Proceedings of the fourth Workshop on Intelligent Solutions in Embedded Systems WISES 06*, pages 1–12, Vienna. ISBN: 3-902463-06-6.

Kaiser, J., Zug, S., Schulze, M., and Piontek, H. (2008). Exploiting self-descriptions for checking interoperations between embedded components. In *International Workshop on Dependable Network Computing and Mobile Systems (DNCMS 08)*, pages 41–45, Napoli, Italy.

MathWorks, T. (2010a). Matlab/Simulink - Website.

MathWorks, T. (2010b). Real Time Workshop - User's Guide.

MathWorks, T. (2010c). Real Time Workshop Embedded - User's Guide.

National Instruments (2009). LabVIEW 2009 - Herstellerseite.

Object Managment Group (OMG) (2003). *Smart Transducer INterface Specification*.

Robert Bosch GmbH (1991). *CAN Specification Version 2.0*. Robert Bosch GmbH.

Schulze, M. (2009). FAMOUSO – Eine adaptierbare Publish/ Subscribe Middleware für ressourcenbeschränkte Systeme. *Electronic Communications of the EASST (ISSN: 1863-2122)*, 17.

Schulze, M. and Zug, S. (2008). Exploiting the FAMOUSO Middleware in Multi-Robot Application Development with Matlab/Simulink. In *Proceedings of the 9th International Middleware Conference (Middleware2008) ACM/IFIP/USENIX*, Leuven, Belgium.

ZigBee Alliance (2003). *ZigBee Specification - IEEE 802.15.4*.

Zug, S. and Kaiser, J. (2009). An approach towards smart fault-tolerant sensors. In *Procedings of the International Workshop on Robotics and Sensors Environments (ROSE 2009)*, Lecco, Italy.