Components @ Work:
Component Technology for Embedded Systems

Peter O. Müller
Christian Stich
Christian Zeidler
ABB Corporate Research
Asea Brown Boveri AG
Speyerer Straße 4
69115 Heidelberg, Gemany
+49 6221 59 [6223|6211|6259]
[peter.o.mueller|christian.stich|christian.zeidler]@de.abb.com

ABSTRACT

During the last years the use of component-based software development (CBSE) gets more and more common for desktop applications to speed up time to market and reduce development costs. Especially in the area of embedded real-time systems the reuse of tested and robust parts of prior applications is of great desire, to fulfill the strong requirements on maturity, availability and cost. But further requirements like low power design and real-time execution of components makes it impossible to use component frameworks well known in the desktop area. This paper discusses the problems of component-based software development for embedded real-time systems and derives requirements for a component framework for this domain. First insights for such a framework are presented.

1. Introduction

Component-based software engineering (CBSE) is quickly becoming a mainstream approach to software development. At the same time there will be a massive shift from desktop applications to embedded systems. More and more traditional IT systems will move from visible desktop computers to invisible embedded computers in intelligent apparatus, e.g. web connected fridges, house automation utilities, PDAs, cellular phones, spontaneous networked devices, just to mention few examples. Furthermore, industrial automation systems become increasingly decentralized, relying on distributed embedded devices (intelligent field devices, smart sensors) to not only acquire but also pre-process data and run more and more sophisticated application programs (control functions, self-diagnostics, etc.). As a consequence of these facts, one can expect that component-based software engineering for embedded systems will be a key success factor for the software industry in the coming decades.

In this paper we briefly out line the specific application domain of CBSE, followed by a scenario example from process automation domain, for which a prototypical implementation is described thereafter. We conclude with our findings in "lessons learned", where we define the needs for effective CBSE and describe a foundation of functionality needed for "embedded CBSE" and an outlook to future activities.

2. Domain Characteristics

But the state-of-the-art in software engineering for embedded systems is far behind other application areas. Software for embedded systems is typically monolithic and platform-dependent. These systems are hard to maintain, upgrade and customize, and they are almost impossible to port to other platforms. Component-based software engineering would bring a number of advantages to the embedded systems world such as fast development times, the ability to secure investments through re-use of existing components, and the ability for domain experts to interactively compose sophisticated embedded systems software.

Visual techniques have been proven to be very effective in specific domains like GUI software composition. Composition of embedded systems software still has a long way to go to reach that level. At the very least, users would benefit greatly from the effective use of visual techniques for providing feedback in the development process (during design, composition, installation, and

during runtime validation). Unfortunately component-based software engineering cannot yet be easily applied to embedded systems development today for a number of reasons. Up to now, the mainstream IT players did not pay much attention to the (so far) relatively small embedded systems market and consequently did not provide it with suitable technologies or off-the-shelf software (such as operating systems or suitable component models). From a technical point of view, these choices were justified by considering the major characteristics of embedded devices, such as limited system resources (CPU power, memory, etc.) and man machine interface functionality, the typically harsh environmental conditions, and the fact that the development and target systems are not the same.

The rapidly changing market makes investment in component-based software engineering for embedded systems not only viable but also essential. The key for industries to benefit from the increasingly powerful and less expensive hardware, is the ability to develop and port embedded software more quickly and at acceptable costs. Vendors of embedded devices would benefit by being able to offer scalable product families, whose functionality could be tailored by flexible composition of reusable building blocks. These families are differentiated by the performance of the hardware and the provided functionality, but are based on re-use of many identical software components. All this requires that the embedded systems software be modular and composed of loosely coupled, largely self-sufficient, and independently deployable software components.

ABB's business unit Instruments develops a large number of different field devices, e.g. temperature-, pressure-, and flow-sensors, actuators, positioners, etc. As the field device hardware becomes more and more commodity, the software determines the competitiveness of field devices. The market demands for additional functionality in shorter time cycles. This means, that software dominates the development and maintenance costs of field devices.

However, today's field device software is monolithic software developed specifically for each field device type. Monolithic software prevents to serve the field device market with value-added features in a cost-efficient way:

- Same functions needed by different field devices are implemented repeatedly at different development locations in different ways (e.g. Fieldbus Drivers, Nonvolatile Memory-Manager, FFT[1]-algorithm).
- Functions and modules are implemented for a specific environment with no standardized interface (e.g. Interrupt-Driven, Port I/O)
- Long development time
- Monolithic software has a fixed functionality that is hard to maintain, to extend, and to customize.

3. Example Scenario: Field Device

The requirements and the architecture of field devices will be discussed in this paper at an example: a pneumatic positioner (TZID), see Figure 1. Pneumatic positioners are used to control pneumatic actuators attached to valves. Main features are:

- low priced and lightweight
- bus powered (2-wire)
- position measurement
- self-adapting position control loop
- diagnostic capabilities
- communication with the control system via fieldbus (Profibus PA and Fieldbus Foundation)

3.1. Requirements of field devices

The following requirements, resource constraints and typical implementation techniques have to be taken into account when discussing a component-based approach for the field device implementation:

Figure 1: Pneumatic positioner TZID

- The available power is only 100 mW for the whole device. This limits the choice of CPUs come into question.

[1] Fast Fourier Transformation

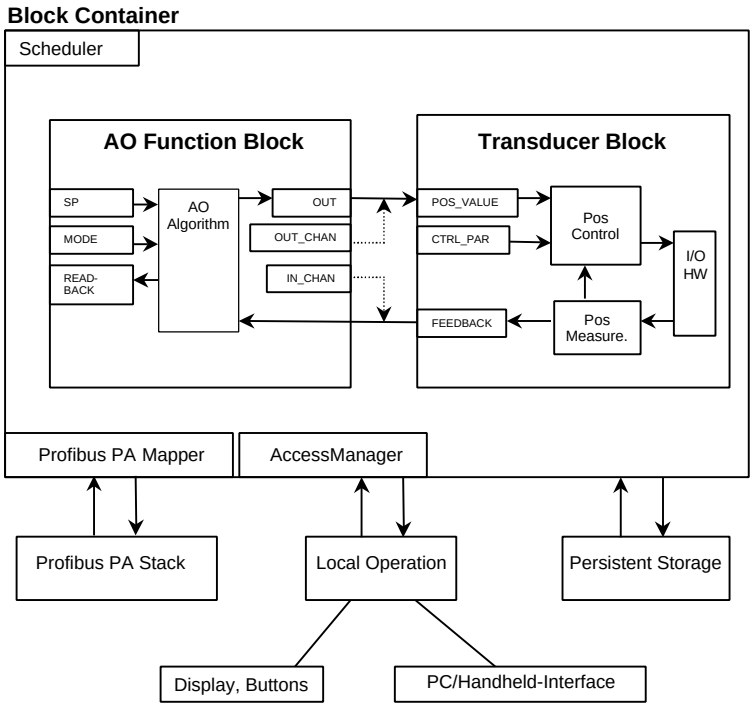

Figure 2: Architecture of a field device

- Software architecture is driven by fieldbus architecture (e.g., function block concept). Fieldbus stacks from 3rd party suppliers are used.
- Parts of the software require real-time execution (control loop, execution of fieldbus function blocks).
- The implementation language today is C. C++ may become an option, possibly in a special dialect like Embedded C++[2]. However, such C++ or EC++ [3] compilers are not available today for the relevant low power micro-controller.
- The device has a static software configuration, i.e. the firmware is updated/replaced completely, no dynamic loadable functionality (this may change in future).
- Many field devices are used in safety critical areas, e.g. chemical plants. Therefore costly certification procedures are required for such devices.
- Typical lifetime of field devices is up to 10 years.

3.2. Component-based architecture for field devices

Figure 2 illustrates a first attempt how to componentise the software for a field device. As stated above, one main driver for this software architecture of field devices is the fieldbus architecture. The example is tailored for Profibus [4].

In the following, the main components, their responsibilities, their composition relations, and the main motivation, why to pack a certain function as component, is discussed.

3.3. Block, Sub Block, and Parameter

From a fieldbus point of view, blocks are the most natural components for a field device. They provide the building blocks of the fieldbus application. In Figure 2, two blocks are shown: an Analog Output Function Block representing the *automation function* of the device within the control system and a Transducer Block representing the *parameters & functions* of the connection to the process (i.e. position measurement and control of pneumatic converter). The interface of a block is defined by its parameters. Parameters represent process and configuration data and have a number of attributes like: the actual value, data type, storage class (constant, dynamic, non-volatile), access rights, default value, parameter type (in, out, contained) etc.

Block and parameter as software components should pro-

[2] EC++ is a subset of C++ omitting templates, exceptions, RTTI, multiple inheritance, etc. in favor of high performance, low memory consumption and ROM-able code; see [EC++99]

vide a model for the implementation of Function and Transducer Blocks with the following features:

- It is independent of the particular fieldbus. This seems to be easy for the two most popular fieldbusses used in process industry (Profibus, Fieldbus Foundation[5]), because they share the same roots.
- It should support a component-based implementation of the block algorithm by sub-blocks. This is especially important for transducer blocks because the same transducer block algorithm should be reusable between a Fieldbus Foundation (FF) and a Profibus device although the transducer blocks themselves may have a different interface in terms of parameters.
- It supports infrastructure functions that are beyond the scope of the fieldbus application like user/factory access control for parameters, persistent parameter storage, and support for parameter display. This support needs to be expressed by additional block/parameter attributes and operations.
- It provides an optimal memory use for the parameters and their attributes (ROM and RAM). This requires support from the component model to specify and implement different memory classes.

3.4. Block Container

The Block Container provides the run-time environment for Blocks. The main idea is to provide an execution model for blocks that relieve the blocks from dealing with the details of scheduling, parameter transfer between blocks, and parameter access synchronization. The responsibilities of the Block Container are:

- Scheduling the execution of blocks. The scheduling strategy also implies a strategy for synchronizing parameter access by different threads (e.g. acyclic parameter access from fieldbus, block execution, sub-blocks having an own thread).
- Handling the parameter transfer between blocks. The fieldbus specifications define different communication mechanisms between Function Blocks (configured by link objects) and between Function Block and Transducer Block (configured by channel parameters), see Figure 2. The Block Container should provide a unified block communication that covers both ways.
- Interfacing to the fieldbus and handling of cyclic (process data) and acyclic (configuration data) fieldbus services that result in parameter accesses.
- Enforcing an access policy for block parameters (read, write, passwords, etc.)

In this approach, most of the work is delegated to the Block Container, so that the Block algorithm itself can always rely on valid parameter values and don't need to take care for synchronization.

The concept of Block Container and Blocks is common to both Profibus and FF. However, adaptation to the used bus is needed in the following areas:

- different Blocks are used (different parameters and block algorithm)
- different scheduling strategies (FF has a system wide block schedule defined by the FF standard, Profibus has not)
- different communication services for process values (FF uses publisher-subscriber model allowing slave-to-slave communication, Profibus uses cyclic data transfer between master and slave only).

Therefore, we suggest to implement functions like block scheduling, parameter access control, and mapping of the field bus stack to the Block Container in separate components that can be plugged into the Block Container as shown in Figure 2.

3.5. Fieldbus Stack and adaptation to Block Container

Fieldbus stacks are standard components, most preferably taken from 3rd party vendors. Adaptation to the Block Container is done via a Mapper component. The main responsibilities for the Mapper are:

- mapping of fieldbus specific addressing schemas to Block and Parameter schema internally used.
- translation of cyclic and acyclic fieldbus services into read/write accesses to Block Parameters managed by the Block Container
- handling of device management (directory objects)
- handling of link objects, i.e. configuration of the inter-Function Block communication at the Block Container level

The functionality of the Mapper (and partly also of the Block Container) depends on the functionality provided by the communication stack. Most stacks are designed in a way that an application must be implemented “around” the stack. E.g. for callbacks one has to add code in partly written functions. This makes it hard to separate the application from the stack. For encapsulation it would be better to have an API that makes it possible to register

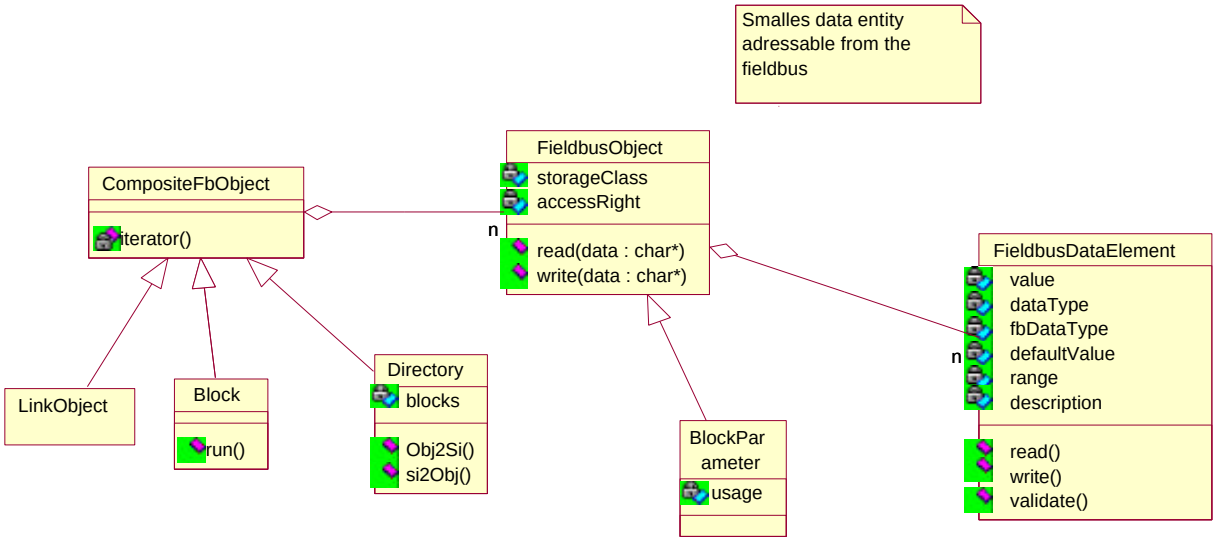

Figure 3: Class diagram showing the objects living in the block container.

callback methods or a handler class [6].

3.6. Local Operation

Local operation includes parameter access and changing them directly at the device using a local display, connecting a Service PC via RS232 or a wireless link. The Block Container should handle local parameter access similar to the acyclic fieldbus services.

3.7. Persistent Storage

Some Block Parameters have to be stored non-volatile, e.g. in EEPROM[3]. The Persistent Storage component is responsible for saving/loading persistent data, managing the programmable memory, error detection and correction, versioning etc.

4. Case Study Implementation

Our scenario depicts just few aspects, which are essential to field device realization. Those are used to implement a framework outlined in this section. Because building of frameworks is hard [7] we have tried to spend as much time as possible to:

- Get an in-deep knowledge of the problem domain.
- Analyze code from the domain to find patterns that solve typical problems.

The framework has a better chance to be successful when several iterations are made. Therefor we have started to develop a first device prototype that will be discussed in the following section. It can be seen as starting point for further design discussions and a first proof of concept.

Figure 3 shows the objects living in the block container. The main object is the Block (function blocks like analog output, transducer block, control algorithm blocks). A Block contains parameters representing the state of the block and an algorithm executed by a run() method. Block parameters must be accessible from the communication stack. Both cyclical and sporadic access has to be supported.

A FieldbusObject is the smallest addressable unit from the fieldbus point of view. Its data type can be a simple type (e.g. integer, float, string, time), a structure of simple types, or an array of simple types. Therefore, a FieldbusObject is modelled as a collection of one or more FieldbusDataElements. A FieldbusDataElement represents a simple data type with additional low-level functions like range checks for data validation.

An example for a FieldbusObject is the set-point parameter of an Analog Output function block consisting of the FieldbusDataElements VALUE and STATUS.

During the first implementation a bunch of open questions came up. The following items show some open issues:

[3] Electrical Erasable Programmable Read Only Memory

Component	Reusability	Variation Points
Block Container	Through all field devices	different Scheduler, Access Manager and Fieldbus Mapper possible for adapting to field bus
Block (concept and interface)	Through all field devices	parameters and their configuration, sub-blocks
Function Blocks	Through all field devices	none, most of them are predefined by fieldbus specs
Transducer Block	only within one device family (profile)	Implementation of transducer block algorithm composed of sub-blocks. Same transducer block algorithm composed of sub-blocks can be packed into FF and PA Transducer Blocks having different interface in terms of parameters.
Local Operation	through all field devices	different implementations (e.g. display/buttons, infrared, bluetooth)
Persistent Storage	through all field devices	different implementations possible for EEPROM or Flash-PROM
Fieldbus Stack and Fieldbus Mapper	through all devices of one bus type	none beside stack configuration (will be a third party component)

Table 1 Possible Components and their characteristics

1. Each time e.g. the value of a FieldbusObject should be read or written the request has to be checked for validity (e.g. valid datatype). In some cases it is necessary to make sure that a change in one FieldbusObject is consistent with the values of some other FieldbusObjects within a block. Then the responsibility for this set of objects could not be in one FieldbusObjects.
2. The implementation of a block algorithm may be active (i.e. have its own thread of control). For active Blocks synchronised access to its FieldbusObjects is required. Several design alternatives exist:
   - One global lock in the container to fine-grained locking in the get/set methods of each Block.
   - Commercial tools like Rhapsody [1] or RoseRT [2] provide an event-based solution (Ports and Protocols) that the user must use for all active objects.
3. For cyclical requests FieldBusObjects from different blocks must usually be used to create a response frame. FieldBusObjects distributed over different blocks requires iteration to create the response frames and therefore some kind of synchronisation.

The proposed architecture divides the field device software into two parts:

- The infrastructure (framework) containing the Block Container, Block and Parameter concept, Scheduler, Access Manager, Persistent Storage, Local Operation, Fieldbus Stack and Mapper, and
- The application dependant Blocks like Function Blocks and Transducer Blocks.

Table 1 summarizes the components, their reusability, and variation points to adapt the components to the specific field device.

The main challenge lies in providing a framework that is reusable within all kind of field devices. Therefore, the architecture and its supporting component technology should provide:

1. A framework that can be easily adapted to the specific field device.
2. An implementation model for Function - and Transducer Blocks that relieves these blocks from infrastructure tasks like execution scheduling, persistent parameter storage, access control, and access synchronization. Furthermore this allows a component-based implementation of the blocks itself to broaden the potential reuse of finer-grained functions within a device family (e.g. different control algorithms).

5. Lessons Learned

The scenario gives indication, which functionality and ability a component model should provide from the point of view of the application domain. To name just a major one, which is to handle resource constrains and it's implications on the component technology.

To cope with the resource limitations is one domain specific problem. Another one is to support development of real-time application assembled out of components. This is a challenge by itself and a topic of investigations of the last tens of years. The most prominent approaches are RoseRT by Rational [2] and Rhapsody by Ilogix [1]. Both of them apply the event base programming style and support implementation based on state automata, but do not consider reuse or component orientation as their major drivers. Therefor they start with UML-like specification and extend the definition tools with functionality that provides code generation for a specific target. Both approaches do not consider neither component model definition nor architectures, beyond the event based communication of “capsules” [2] or “active objects” [1]. Composition of applications out of components and active reuse support by appropriate repository implementation is not offered adequately either. For more details on that state-of-the-art tools see the appropriate section in this report.

In order to make component-based software engineering happen, not only for field devices, and to achieve a reduction of development cost and time by reuse of established and proven components, it is not enough to solve only one of the presented obstacles. An overall approach for the development of component-based embedded software is needed.

As we believe this approach has to comprise several main features as depicted in Figure 4, which we have categorised in five groups and describe below. In a first outline the identified groups should concentrate of the following issues:

Component model:

- addresses non-functional properties and constraints such as worst-case execution time and memory consumption
- allows to specify efficient functional interfaces (e.g. procedural interfaces)
- allows to specify architectural styles that describe components connections and containment relations
- allows for code generation and controlled component adaptation when architectural styles are applied to components (source language or generative components)

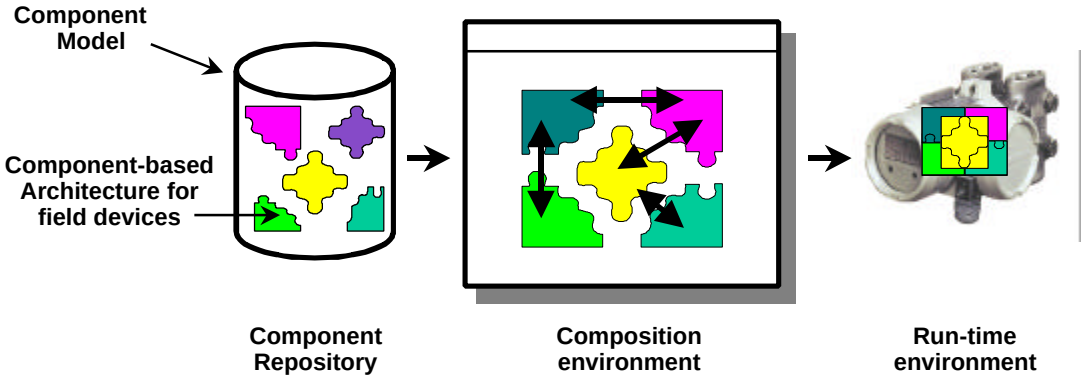

Figure 4: Component technology for embedded devices

Component-based architecture for field devices:

- a framework for field devices that is expressed as standard interfaces, components, and architectural styles
- is based on field bus architecture
- express compile-time optimization abilities, which could be applied during target code preparation

Repository:

- storage and retrieval of components during analysis, design, implementation, and composition
- stores components and architectural styles according to the component model including interface descriptions, non-functional properties, implementation

(potentially for different micro controllers), support scripts for composition environment, test cases
- supports component versioning

Composition Environment:

- supports composition techniques (visual or script based)
- checks composition rules attached to architectural styles in order to verify that a component configurations meets their constraints
- performs component adaptation and code generation for the application
- supports definition of composition rules, which in an subsequent step could be compiled to architectural styles description

Run-time Environment:

- provides an efficient implementation model for components
- addressing the constraints for field devices: low available memory, implementation possibly necessary in C or optimized C++
- supports the approach to compile a component-based design into a optimized firmware for the embedded device, thus having no run-time environment beside the RTOS
- allows for a hardware and RTOS independent implementation of components (e.g. by an RTOS abstraction layer)

Based on these five categories, which make up the major ingredients for a component-based systems (CBS) development, we outline a vision of a software development process taking these considerations into account.

6. Conclusion

In order to bring the advantages of component-based software engineering to embedded systems the special domain characteristics have to be taken into account. To manage the development of the components a comprehensive proficient approach for embedded software is needed. We divided this process into five categories in which is concentrated to deal the requirements of field devices.

To build a case study for an usable component based software engineering system for field devices the ability must be given to retain the special needs and requirements of the field devices in each step. We will address this aspects in future work in context of the PECOS [8] projects.

REFERENCES

[1] http://www.ilogix.com/fs_prod.htm
[2] http://www.rational.com/products/rosert/index.jsp
[3] Embedded C++ Specification, http://www.caravan.net/ec2plus, 1999
[4] Profibus omepage, www.pno.org
[5] Fieldbus Foundation homepage, http://www.fieldbus.org/
[6] D. Schmidt, I. Pyarali, The Design and Use of the ACE Reactor, Washington University, http://www.cs.wustl.edu/~schmidt/ACE-papers.html
[7] F. Buschmann, A. Geisler, T. Heimke, C. Schuderer: Framework-Based Software Architectures for Process Automation Systems, Proceedings of the 9th IFAC Symposium on Automation in Mining, Mineral and Metal Processing (MMM '98), Cologne, Germany, 1998
[8] PECOS Project Web Site, http://aurora.unibe.ch:8080/