PECOS - Pervasive Component Systems

Peter Müller, Christian Zeidler, Christian Stich
peter.o.mueller|christian.zeidler|christian.stich@de.abb.com
ABB Forschungszentrum Heidelberg

Andreas Stelter
andreas.stelter@de.abb.com
ABB Automation Products GmbH

13th March 2001

Abstract

The PECOS project aims to enable component-based software development for embedded systems, specifically for field devices. The project addresses the major technological deficiencies of state-of-the-art component technology with respect to embedded systems by developing (I) a Component Model for embedded system components addressing behavior specification and non-functional properties and constraints, (II) a Component Repository utilizing this model, supporting a composition environment and interfacing to a component specification environment, (III) an interactive Composition Environment for composing embedded applications from components, validating functional (e.g., interfaces) and non-functional compositional constraints (e.g., power-consumption, code size), generating the application executable for the embedded device and monitoring their execution, (IV) an Ultra-light Component Environment to install, run, test, and tune component-based applications on resource limited embedded systems and enable their management.

In this paper, we discuss the requirements of field devices with focus on resource constrains and it's implications on the component model, the composition environment, and the run-time

environment. PECOS is an EC project starting in September 2000. The participants are ABB, OTI Netherlands, Forschungszentrum Informatik Karlsruhe, and University of Berne.

1 Introduction

ABB develops a large number of different field devices, e.g. temperature, pressure, and flow sensors, actuators, positioners, etc. The market demands for additional functionality in field devices like asset management support, diagnosis, and seamless integration into automation systems. This also means, that software will dominate the development and maintenance costs of field devices. However, today's field device software is mostly monolithic software, developed specifically for each field device type. Monolithic software makes it hard 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. adapters to communication stacks, persistent memory manager, control algorithms).
- Functions and modules are implemented for a specific environment with no standardized interface.
- Long development time
- Regression-Tests after software modification are often large scaled because of non deterministic side effects.
- Monolithic software has a fixed functionality that is hard to maintain, to extend, and to customize.

Component-based software engineering can bring a number of advantages to the embedded systems world such as (I) faster development times, (II) the ability to secure investments through re-use of existing, well tried, components, (III) the ability for domain experts to interactively compose embedded systems software and to adapt the software to specific customers needs.

The state-of-the-art in software engineering for embedded systems is far behind other IT application areas. This is especially true for embedded devices with hard resource constraints such as field devices.

2 Requirements for Field Devices

The requirements of field devices will be shown at an example: a pneumatic positioner (TZID)

Figure 1: Pneumatic Positioner TZID

as developed by ABB (see Figure 1). Pneumatic positioners are used to control pneumatic actuators attached to valves.

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:

- The available power is limited from the fieldbus physical layer specification.
- Software architecture has to fulfill the fieldbus architecture (e.g., function block concept).
- Parts of the software require real-time execution.
- The implementation language today is C; C++ may become an option, possibly in a special dialect like Embedded C++. However, such C++ or EC[1]++ compilers are not available today for relevant microcontrollers.
- The device has a static software configuration, i.e. the firmware is updated/replaced completely, no dynamic loadable functionality (this may change in future).
- Field Device has to provide three main areas with increasing functionality: Local User Interface, Fieldbus, Process

3 Component-based architecture for field devices

This section outlines our first attempt how to componentize the software for a field device. As stated above, one main driver for this software architecture is the fieldbus architecture. The example is based on a TZID for Profibus PA. In the following, the main components, their responsibilities, their composition relations, and the main motivation, why to pack a certain function as component, are discussed.

[1] 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 [EMC++]

3.1 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. As an abstaction for a simple device we took an Analog Output Function Block representing the automation function of the device within the control system and a Transducer Block representing the parameters and 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, nonvolatile), access rights, default value, parameter type (in, out, contained) etc. Block and Parameter should provide a model for the implementation of Function- and Transducer Blocks with the following features:

- It is independent of the particular fieldbus. This may be possible for Profibus PA and Fieldbus Foundation (FF) because both busses have a similar block model (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 FF and a Profibus PA 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 access control for parameters or persistent parameter storage.
- 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.

3.2 Block Container

The Block Container provides the run-time environment for Blocks. The main ideas are (I) to provide an architecture for the field device application that is independent of the used communication stack and (II) to provide an execution model for blocks, that relieves 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.

- 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.)

3.3 Summary

The proposed architecture divides the field device software into two layers: (I) an infrastructure layer consisting of Block Container, Block and Parameter concept, Scheduler, Access Manager, Persistent Storage, Local Operation, Fieldbus Stack and Mapper, and (II) an application layer containing Function Blocks and Transducer Blocks.

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

4 Implications on component technology

While component technology promises an escape from monolithic software that is expensive and hard to maintain for common purpose IT solutions, it is questionable if it does too for programming of embedded devices. Moreover it is unlikely that state-of-the-art component technology like COM, Corba, JavaBeans/EJB and component technology as it is currently discussed in literature (e.g., [4], [1]) can be applied as is to field devices. In the following, aspects of component technology are discussed in the context of the requirements for field devices. We start with the component definition postulated by Szyperski [4]:

> *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

4.1 Contractually specified interfaces

State-of-the-art component technologies specify interfaces as pure collection of methods (events

Component	*Reusability*	*Variation Points*
Block Container	Through all field devices	Used Scheduler, Acess Manager
Block (concept and interface)	Through all field devices	-
Function Blocks	Through all field devices	-
Transducer Blocks	Within one family of devices	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	Display Type, Conneciton (local, remote)
Persistent Storage	Through all field devices	Variations because of different persistent memory types
Communication Stack and Mapper	Through all field devices using one bus type	-

Table 1: Components and their expected re-usability.

and attributes are finally modeled as interface methods as well). However, for embedded software and especially embedded real-time software, non-functional specifications like memory consumption of a component, worst-case execution time of a method, and expected power consumption of a component under a certain execution schedule are an equally important part of the contract. The first declare which functionality is provided the later how it is provided, which express the non-functional part of the semantic behavior of this method execution. The currently progressing UML profile for scheduling, performance, and time [2] may provide a specification means, which does the first step towards expressive interface declarations. For example, having specified the worst-case execution time of function blocks would allow the verification that the overall schedule of all function blocks can be met.

Component interfaces are usually implemented as object interfaces supporting polymorphism by late binding. While late binding allows connecting of components that are completely unaware of each other beside the connecting interface, this flexibility comes with a performance penalty. A component model for embedded devices should allow for procedu-

ral interfaces, object interfaces with and without polymorphism. Procedural interfaces can be used for stateless component instances and component singletons. Object interfaces without polymorphism can be applied if the target component implementation can already be determined at design time. Such optimization gain in a lower calling overhead that sums up in the case of fine-grained components.

Semantic specification like pre- and post-conditions are of great value for the software quality especially if they are checked during run-time. However, for embedded devices these additional run-time checks may turn out not to be feasible due to the limited CPU resources. An approach could be to distinguish between debug and release versions of a component (similar to C++ assertions). The release version runs without run-time checks meeting the performance and power consumption requirements. The debug version runs with run-time checks enabled requiring higher CPU clock (= higher power consumption). Alternatively, design-time checking using a composition environment which either simulates or calculates the correctness of connected components with given pre- and post-conditions could be a viable way.

4.2 Unit of composition and independent deployment

State-of-the-art component technologies allow for component composition at design-time and at run-time. Both [4] and [1] take the position that components are binary units of deployment that should be deployable to a component system at run-time. To fulfill these requirements, support from the component model (e.g. late binding), support from the run-time environment (e.g., life-cycle management, dynamic loading, garbage collection or reference counting), and dynamic communication mechanisms like the JavaBeans' events or COM connection points are needed.

Having the low resources of field devices in mind, we argue that such a run-time infrastructure is too expensive in terms of processing power. Embedded devices of the discussed class can not afford the overhead of garbage collection or reference counting, the overhead of late binding for every interface method especially for fine-grained components, and the memory overhead required for the infrastructure itself and for each component needed to support the infrastructure. Therefore complete stripping of this functionality or sensible degradation is required.

In addition, design-time composition allows for optimization: in a static component com-

position known at design-time, connections between components could be translated into direct function calls instead of using dynamic event notifications. Such optimizations probably require components to be available in source code or at least introspection abilities at design-time. Composition tools are required that can inspect and adapt such components. On the other hand, source code components can provide support for composition tools in form of meta-information and scripts to be executed by the composition tool.

Finally, design-time composition could be the instance of specific adaptation of components and generated code towards specific microcontroller families and RTOS APIs [3].

4.3 Explicit context dependencies

Beside other interfaces and components that are required for a component to work, context dependencies also include the required run-time environment such as CPU, real-time operating system, and component implementation language (with respect to the binary interface). From the viewpoint of a state-of-the-art component technology, this run-time environment can become quite basic for embedded devices due to the resource and real-time constraints. Beside JavaBeans, component models provide programming language independence by a binary object model or by different language bindings. We argue that abandoning programming language independence in favor of higher performance is acceptable for embedded devices. In the case of source language components as discussed in 4.2, the composition support in form of meta-information and scripts to be executed in the composition environment comes in as additional context dependency.

4.4 Reuse

Black-box component reuse seems to be the best solution since it hides component implementation completely from the client. Source language components as discussed in 4.2 require to open parts of their implementation leading to grey-box or even white-box reuse. According the [4], grey-box and white-box reuse very likely prevents the substitution of the reused component by other components. However, establishing clear conventions about the available knowledge of the implementation and the allowed changes of the implementation should help to overcome this problem. If this knowledge can be captured completely in architectural styles (e.g. component connectors) or in composition scripts belonging to the component, one could reach a grey-box reuse from the composition environment's point of view but a black-box reuse from the component user's point of view.

Only the composition environment would be allowed to use knowledge about the component's implementation.

4.5 Portability, platform independence

The demand for reusable software components directly leads to the requirement of platform independence and portability because software components as abstractions of application functions will have a longer lifetime than the hardware and the used microcontroller. But that leads directly to an either conceptually provided abstraction layer in terms of programming standards or an implementation based solution like virtual machine.

We argue that source level portability will be sufficient (or better: must be sufficient). Source level portability requires agreement on the implementation language (e.g. ANSI C or C++). Microcontroller specific language extensions provided by many compilers for the embedded domain prevent source level portability, require manual porting effort, and finally lead to a version explosion of the component. Source level portability also requires agreement on the available libraries like ANSI C run-time library, operating system API, hardware access, device drivers etc. One way to achieve this is to provide proper abstractions e.g. for the RTOS API, that are specified according to the used component model.

Binary platform independence as provided e.g. by the Java platform is not a requirement for the discussed class of field devices. However, this may change in the future if run-time component composition and deployment will become a requirement. Also, from a development productivity perspective, it would be very desirable to have a Java platform available.

4.6 Component wiring (scripts, connections)

Component wiring gets an emphasized role. Once the components are present, efficient and flexible composition of new application out of existing components gets the first priority. Therefore, composing an application in drag & drop manner, while preserving the consistency of the new composed application pops up as one of the most challenging things. On one hand it requests an advanced component model and at the same time support for expression of architectural styles in order to provide prescription how to construct according to given domain rules. On the other hand it gives the ability to optimize component interaction by source code adaptation or interweaving of component

glue code. For performance reasons, we argue that component glue code has to be generated in the implementation language (e.g. C or C++). Script languages as glue code known form state-of-the-art component technologies won't be affordable.

4.7 Conclusion and direction

In this paper we have discussed at the example of a pneumatic positioner the requirements of field devices with focus on resource constrains, a possible component-based architecture for field devices, and it's implications on the component technology. In order to make component-based software engineering happen for field devices and to achieve a reduction of development cost and time by reuse of 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.We believe this approach have to comprise several main features as depicted in Figure 4, which we have categorized in fife groups and describe below.

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

Component-based architecture for field devices expressed in a framework specifying standard interfaces, components, and architectural styles.

Repository for (I) storage and retrieval of components during analysis, design, and implementation, (II) storing components and architectural styles according to the component model including interface descriptions, non-functional properties, implementation, support scripts for composition environment, test cases and (III) support for component versioning.

Composition Environment supporting (I) visual composition techniques, (II) checks composition rules attached to architectural styles in order to verify that a component configurations meets their constraints and (III) performs component adaptation and code generation for the application.

Run-time Environment providing (I) an efficient implementation model for components, (II) addressing the constraints for field devices stemming from low-power design and real-time execution and (III) support the approach to compile a component-based design into a

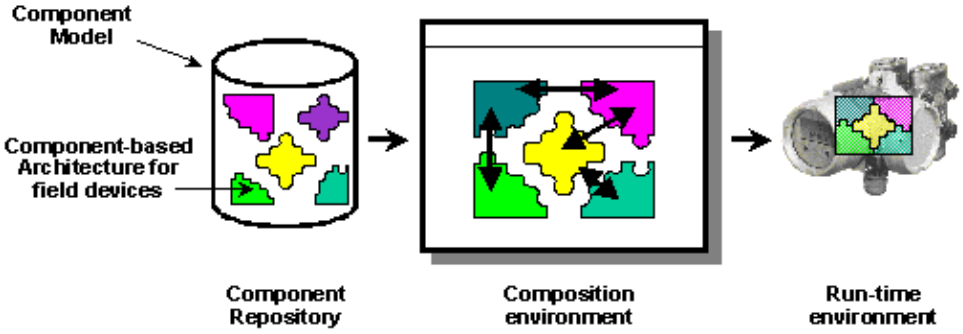

Figure 2: Component technology for embedded devices

monolithic firmware for the embedded device, thus having no run-time environment beside the RTOS.

PECOS, a 2 year EC project starting in September 2000, addresses these issues and will prove this concept by a case study.

References

[1] F. Griffel. *Componentware: Konzepte und Techniken eines Softwareparadigmas*. dpunkt-Verlag, 1998.

[2] Object Management Group. Uml profile for scheduling, performance, and time - request for proposal. *OMG document ad/99-03-13*, 1999.

[3] Jonathan Larmour Simon Fitz Maurice. Source level configuration in embedded systems. In *Embedded Systems Conference Fall*, 1999.

[4] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1998.