The PECOS Software Process

Michael Winter[2], Christian Zeidler[1], Christian Stich[1]

PECOS is a collaborative project between industrial and research partners that seeks to enable component-based technology for a class of embedded systems known as "field devices". In this paper we introduce to the software process, which we are currently developing within this project. In particular, we address the management of non-functional requirements and constraints, as well as architectural issues and the idea of product lines. We report on the current status of the PECOS process, including its application within an industrial case study.

Component-based Software Engineering, Software Process, Component Technology

I. INTRODUCTION

Software engineering is far less developed for embedded systems than for other application areas. Software for embedded systems is typically monolithic and strongly platform-dependent. Embedded Systems are difficult 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 shorter development times, the reuse of existing components and architectures, and the ability for domain experts to interactively compose and adapt sophisticated embedded systems software.

Unfortunately, the mainstream IT players have not paid very much attention to the (so far) relatively small embedded systems market, and consequently little component-based technology or tools for embedded systems exists. The PECOS Project [6][7] is aiming at enabling component-based development for embedded systems, taking the domain of field devices as a case study. The central concept of PECOS is a component model, which is suitable for embedded devices. This model is supported by the COCO language, which serves to specify components and system architectures. For an overview of PECOS concerning the component model and the CoCo language see [1].

[1] ABB Corporate Research
Wallstadter Strasse 59
68526 Ladenburg, Germany
Tel.: +49 (6203) 71-6251, Fax: +49 (6203) 71-6253
[Christian.Stich|Christian.Zeidler]@de.abb.com

[2] Forschungszentrum Informatik (FZI),
Haid-und-Neu-Straße 10-14,
76131 Karlsruhe, Germany,
Tel.: +49 (721) 9654-636, Fax: +49 (721) 9654-637
Email: winter@fzi.de

Around the model we are working on a development environment (the composition environment), which integrates a number of tools for efficient software development, e.g. a tool for schedule generation. But this is only half of the truth. The best tools will not help much, if they are not employed systematically. A clearly defined development process is needed in order to get as much out of the component oriented approach as possible. The software process has to answer the question, which has to perform which development tasks at which point in time; what the necessary input for this task is and which artifacts it is intended to achieve. In order to enable a reasonable development for field devices, we are currently evolving a suitable software process within the PECOS project, which will be presented in the following sections of this paper.

Therein, we want to focus on two central issues. The first question we try to answer is how we enable the development of families of PECOS field devices that means families of field devices, which are somehow similar and rely on a common architecture. This leads us to the question how architectural patterns for field devices can be specified and how we develop a specific device from a predefined architecture. The second issue concerns the composition a PECOS field device from an asset of pre-fabricated components and how these components have to be coupled not only on the functional, but also on the non-functional level. This concerns first of all the development of global schedules for specific field devices, but also the topic of design rule checking.

The organization of the rest of the paper is the following. In section III we present a high-level overview of a component based software development process for PECOS field devices. We then describe the tasks for component development in section IV. Then, in section V we present in detail the tasks that have to be performed during application composition, with a focus on architectural issues and the treatment of non--functional requirements. Finally, we conclude with a summary and outlook of work to be done next.

II. SOFTWARE PROCESSES

Software processes span the development, use, and evolution of software systems. These processes consist of a partial ordering of tasks, decomposable into sub-tasks and actions that collectively describe how software systems come to be the way they are. People with various skills and resources perform these processes using a variety of automated, semi-automated, or manual tools and techniques. Over the time many different software

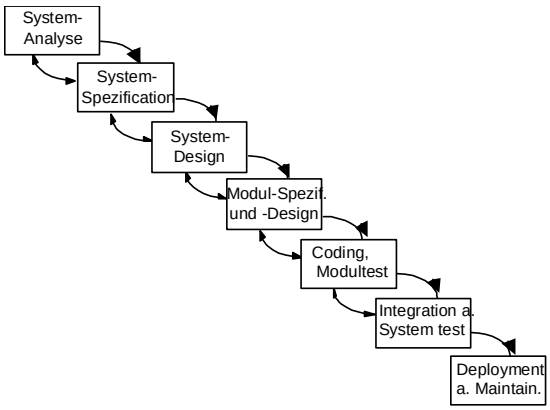

Figure 1 Waterfall Model

development processes where developed and are used in different organizations.

Activity oriented: **Waterfall model**: A model of the software development process in which the constituent activities, typically a concept phase, requirements phase, design phase, implementation phase, test phase, and installation and checkout phase, are performed in that order, possibly with overlap but with little or no iteration, see Figure 1. Major drawback of this approach it the strict sequential work order, which results in stringed workflows and high risks and costs in case of failure detection e.g. specification errors etc.
A specific flavour of that approach is the V-Model, which defines the counterparts of activities, which have to be performed in order to enable the validation of the system with respect to the specification.

Phase driven Models: Other developed approaches concentrate on the development phases and order them in different relations in order to optimize iterative development approach or enable parallel development on a software product. Well known Approaches like the **spiral model** of Boehm [5], see Figure 2, and some new approaches as the **Rational Unified Process (RUP)** [3], see Figure 3.

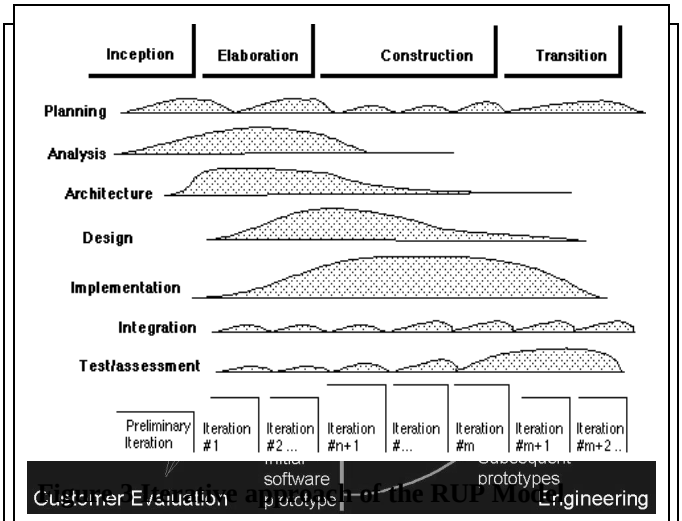

Figure 2 Spiral Model of Boehm

Common to all approaches it the introduction of incremental development style to manage the complexity reduces risks and introduces a more flexible project planning. It's worthwhile to read the referenced books in order to get into more details.

III. PECOS Process Overview

The PECOS process aims to enable component-based software development for embedded systems, specifically for field devices. This process addresses the major technological deficiencies of state-of-the-art component technology with respect to the non-functional requirements of embedded systems, such as limited CPU power, memory and hard real-time. The goal of the PECOS project is to consider this issues more thoroughly in order to enable assessment of these properties during the construction time.
We distinguish two kinds of development, the application and the component development.

The topic of software processes with respect to components concentrates on the methodology of software development for a selected component model (e.g. EJB, COM, CORBA Component Model or even self-made like PECOS one). It defines a development processes or selects one of the existing, e.g. RUP [3], Catalysis [4] etc., and adopts or extends them to the needs of what is missing or needed for CBSE. It has to be considered for initial system development as well as for evolving software systems, the latter comprising to some extend re-engineering aspects. In general all software development phases have to be reconsidered in context of software components.

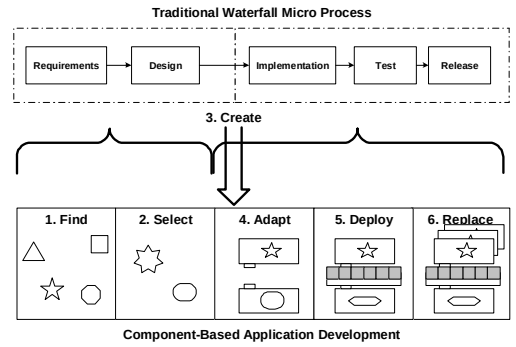

Figure 4 CBSE Process: 2 layers

The two listed examples could serve as a blueprint for the software processes definition, even if they are matters of religious discussions. What we indeed try to outline is a generous way of software process for CBSE. It is of course pretty coarse-grained but enables a common understanding of needed procedures. Thereby we assume that any component-base application development applies an iterative development model. Its n-iteration respectively uses the waterfall model, where requirements specification, rough and fine design, implementation, documentation, testing (module and integration), and getting in operation is executed, see Figure 4.

IV. Component Development

Typically the development of a PECOS component is triggered during application composition when the need for a component is identified and this component cannot be provided from the component repository.

A person in-titled component developer carries out the development of a new PECOS component. This role stresses the distinction between component development and application composition from components.

The tasks of component development, as shown in Figure 5, are presented in the following sub-sections in a linear style. In praxis, several iterations over this process may be necessary.

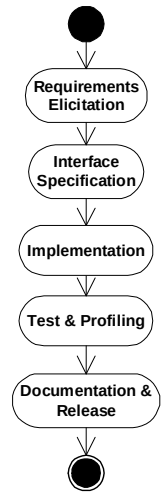

Figure 5 Component Development Steps

A. Requirements Elicitation & Analysis

During requirements elicitation, the requirements of the component are collected. They comprise functional as well as non-functional aspects. The functional aspects concern the data flow in and out of the component. The non-functional aspects concern timing constraints, memory consumption and the like. The application developer who identifies the need for a component with certain characteristics generally performs requirements elicitation.

During *requirements analysis*, the component developer, as formulated by the application developer, reviews the component requirements. Requirements analysis aims at revealing inconsistencies and ambiguities in the requirements, which are inherent to specifications in natural language.

B. Interface Design

The next step is to define the component's interface in a formal way. In PECOS this is done with the CoCo language. The component interface contains information both about functional (data ports) as well as non-functional characteristics (properties) of the component.

1) Specify component type

First, a decision upon the kind of the component has to be taken. The PECOS component model defines three kinds of components: *passive*, *active* and *event* components. For a detailed presentation of the different kinds of components, please refer to the PECOS component model [1]. Then, the component can be characterized by specifying its *name*, *ports* and *properties.*

2) Specify Ports

The component's *in-* and *out-ports* specify the functional part of the interface. PECOS component port is a data exchange point, which is used to communicate with other components. A component may read form its in-ports and write to its out-ports when it is scheduled. The design of the component's ports has to be suitable with respect to the functional and non-functional requirements that have to be met by the component.

3) Specify Properties

In addition to its type and its data ports, a component is characterized by a set of *properties.* Properties are name-value pairs, which describe (typically non-functional) characteristics of a component. Common properties are: memory consumption, worst-case execution time etc. During interface design, the kinds of properties of the component are specified while the property values are typically assigned later in the development process.

C. Component Implementation

The next step to take is to specify the behavior of the component. Therefore the following two tasks have to be carried out.

1) Select Implementation Language

First, an implementation language has to be chosen. At the moment PECOS is supporting C++ and Java. This step may be pre-determined, as the implementation language within a single PECOS application has to be consistent. Thus, the implementation language may be determined in the component's requirements.

2) Generate Code Skeletons & Fill in Code

Depending on the implementation language, a tool is used to generate code skeletons for either Java or C++ classes.

These classes provide a base class with method interface, which is declared as virtual. A derived class has to be implemented to perform the functional behavior of the required component. Each type of component - passive, active or event - requires a different method-interface.

D. Testing

Testing is an important concern to assure the software quality. The level of software testing dealt with in particular is the 'unit test' level and concerns the functional properties of software components.
Testing needs to be carried out in such a way that it is reviewable, (automatically) repeatable, and leaves an auditable trace of what happened. Testing should also be carried out in such a way that these tests can be modified and reused at a later date.

The task of testing consists of the following activities:

- Specify test cases
- Implement test cases
- Execute test cases
- Analysis Test Result

First, test cases have to be specified. Test cases in PECOS specify the relationship between in- and out-port values: A test case describes which results are expected on a components out-ports, for a given setting of the in-ports.

- Inputs to a component under test are parameter values going in the component.
- Outputs are out-port value parameter

Separate test cases are repeated as many times as necessary. Test case implementation is the task of coding the test cases in a form, which enables an automated execution by some testing tool.
PECOS supports three different methods for testing:

- STUBS is an option to simulate calls using so-called 'stubs' and it gives the possibility to simulate the behavior of external calls.
- NEGATIVE checks ensures that the software under test does not do what it should not do.
- TIMING checks. Often it is not only necessary to ensure that a given component has the correct logical behavior. It must be ensure that the timing performance is acceptable.

Test case execution denotes the actual execution of the test. Finally, the test results have to be interpreted and either led to the insight, that a component indeed behaves as expected or that some error has occurred.

E. Profiling

The profiling step collects detailed non-functional information about the component. The goal is to provide explicit reports that indicate for instance how much a particular function takes to execute as well as how often it is called within a component. Runtime performance is collected by a system running remotely the embedded component. This allows manual optimization of the nonfunctional behavior.
Profile collection in a real-time domain is a two-phase process:

a. Online sample collecting
b. Offline profile analysis

This separation allows minimizing the on-line overhead and the time consumption of profile collection by deferring as much work as possible until off-line processing.
We distinguish three different techniques for collecting the non-functional properties in the PECOS process for each component type:

1) Time sampling

The processor's instruction counter is used to monitor and collect timing data.

2) Source code insertion

The profiler uses instrumentation code inserted into the component to determine the non-functional information.

3) Binary instrumentation

The profiler analyzes binary sequences and a new binary containing instrumentation code is generated.

F. Documentation & Release

The component documentation is basic for applicability of the component. The application developer, who reuses the component, needs detailed information about at least the black-box behavior. This component profile has to match his requirements.
Hence, before the component can be released, it has to be properly documented. Documentation is an important subject to increase the re-use and the quality of the component.
Component documentation templates guaranty common description and consist of the descriptions for:

1) Behavior
2) Interface, In- and Out-ports
3) Test result
4) Profile with non-functional properties
5) Recipes and use cases

Meta-techniques help to get the white-box and the behavior description more in detail. Recipes and use cases give ideas how to use the component in composed applications and steer the component developer to have the re-use of the component in mind.
The last step of the component development is to release the component to the application developer. This task has to assure that the component is packaged in a form to be easily used by the application developer. To collect the profile of the component a common test-bed is used in which the component is executed on the target system.

V. Application Development

In the previous section, the development of PECOS components has been presented. As already mentioned, the development of PECOS components is triggered during application composition. This task is the major activity during application developing.

As a starting point for the composition of a PECOS application, a global architecture is given (see section V.A.2). It defines the de-composition of the application into components, which are sub-systems at this level of granularity. This decomposition is stepwise refined and filled with content during application composition.

It has to be pointed out, that application composition is a highly creative activity of system decomposition and that the tasks presented in the sequel are strongly interrelated. Therefore, a purely linear process description like in the waterfall model is surely over-simplified.

In the PECOS process, see Figure 6, just as in any modern software process, the principle of step-wise refinement is addressed by repeated iterations over a linear process. This process is presented in the following sub-sections.

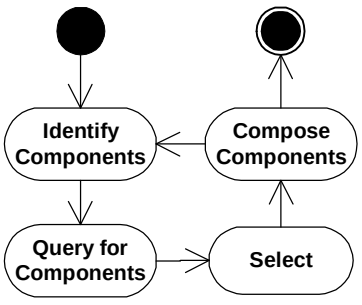

Figure 6 Application Composition Tasks

A. *Preamble*

The requirements specification of a field device addresses the three parts of the device: the mechanics, the hardware and the software. *Application development* is the process of producing a software system conforming to the software requirements of the device.

The requirements specification of a field device addresses the three parts of the device: the mechanics, the hardware and the software. *Application development* is the process of producing a software system conforming to the software requirements of the device.

1) *Requirements Elicitation*

Requirements elicitation is the first activity in every software development process. Nevertheless, it is not addressed within this paper. But we assume that the requirements for the application to be built are readily available. This is legal for the special domain of field devices we are considering here:
In these domain requirements engineering only makes sense for a complete field device, comprising its mechanical, hardware and software parts altogether. As we solely consider the software part here, we may assume the software requirements to be given.

2) *Architecture Specification*

The second step in developing a PECOS application is to design the high-level system decomposition that means the architecture of the application. Typically, the system architecture will not be designed from scratch. But rather a standard architecture is taken and adapted to the needs of a specific PECOS application. This comes from the fact, that field devices are most probably part of a product family with similar requirements and architectures.
The used of a standard architecture has several advantages. First, it promotes a faster development, as high-level design information is reused. And also the probability that ready available software components built for former applications fit into the application is increased.
Second, a standard architecture, which has already been used for several times in slightly modified ways, is well known and has been approved to fit the needs of field devices.

B. *Identify Components*

The first task of each iteration taken during application composition consists of identifying components. On the uppermost level, the application architecture defines the decomposition of the application into components. This decomposition is very coarse-grained and the identified sub-systems generally cannot be realized directly. But they have to be decomposed into smaller components in order to manage their complexity.
The aim of component identification is to decompose a given component into more fine-grained components. This is performed until the establish decomposition is fine-grained enough to be realized.

C. *Query for Components*

The *query for components* task is employed in order to search for ready available components which can be used to realize a given system decomposition. This task is supported by a *component repository*, which contains a collection of available components.
Several concrete techniques can be used to find components in the repository. E.g. a simple string-matching search can be performed. Another possibility is to structure the repository in a tree-like manner and to use this structure to look for components. In praxis, most probably a combination of these two methods will be used.
It is obvious that *querying for components* and *identifying components* are two highly related tasks. The system decomposition (top-down approach) results in components which are needed to realize the application. The other way round, the availability of certain components (bottom-up approach) may influence the system decomposition.

D. *Select Components*

Querying the repository results in a list of available components. These have to be inspected in order to find out, if they can be used to realize a part of the application. Depending on the available components, a number of different actions may be necessary.

If the set of components, resulting from the query, contains a component, which is suitable to directly realize a part of the application, perfect reuse is enabled. This 100% match scenario may not seem very probable. But in the context of product families that rely on a standard architecture, this becomes possible.
In the case of an almost suitable component, adaptation to the reuse context may be possible with little effort. Or, as an alternative, it may be possible to use the component after slightly modifying the system decomposition itself.
If no suitable component can be found, the task of component development is triggered. This case has already been looked at in section IV.

E. Compose Components

The previous tasks are assumed to lead to a set of components, which are suitable to realize (parts of) the application. The next step to be taken is to compose them.
Wiring components in PECOS means to determine the data and control-flow dependencies between these components. It is sufficient to connect corresponding in- and out-ports.
The control-flow is responsible to actually push the data through the connections. This is addressed by an application-global schedule, which conforms to the timing requirements of the device. For a detailed presentation of the PECOS execution model see [6].
Composition rules and contracts are used to specify constraints over a composition and to back a correct-by construction approach. Rules designate constraints over a component or composition in terms of predicates over component properties. They only refer to statically available information, as they do not direct to information that is only attainable at runtime.
Besides determining the constraints and the flow between components, additional non-functional properties have to be taken into account. One example therefore is power consumption.
After composing the composition compiler maps active components to separate threads. The assignment of priorities, periods and deadlines are specified by component properties. The application schedule is computed form these values.
To be able to execute the composed application an execution environment is introduced. A very thin abstraction layer abstracts from the underlying real-time operating system and provides language independent interfaces for synchronization and a common application-programming interface.

F. Application Test

The non-functional and the functional behavior of the composed application have to verify according to the application requirements.
The task of testing consists of the following activities:

- Specify test cases
- Implement test cases
- Execute test cases
- Analysis Test Result

First, test cases have to be specified. A test case describes which results are expected.
Test case implementation is the task of coding the test cases in a type, which enables an automated execution by an application-testing tool.
Automation is used to replace or supplement manual testing with a suite of test programs. Manual testing often yields inconsistent coverage and results. An automated test suite ensures the same scope and process is used repeatedly each time testing is performed. The automated test detects functional and performance issues more efficiently, allowing focusing on improved time to market, repeatable test procedures, and reduced testing costs.
Every integration test should include a test plan and a test report. The test plan includes a detailed description of the test scope, setup and procedures. It is essential when the test requires manual operation. It should also include a trace-ability matrix with the functional specifications. The test report should be issued and updated after major testing sessions (like a new release) and contain indication of all problems encountered during testing [8].
Functionality testing verifies that all of an application's features perform in a fitting manner. Application designers usually perform functionality tests while designing and debugging single components or the whole application.
Reliability testing ensures that the application functions properly. To perform reliability testing, the application runs it continuously for several time, error are monitored the entire time.
For new application versions is our aim to have fully automatic regression testing for integration. Regression testing evaluates the performance and functionality of software and software upgrades. Regression testing differs from functionality testing because it tests features that the hardware or software carries over from its previous version, rather than testing new features. Regression testing demonstrates how a new version of a composed application fixes bugs from the previous version and finds bugs that the new version introduces.

Finally, the test results have to be interpreted and either guide to the insight, that the composed application in fact behaves as expected or one more application assembly iteration step is needed.

G. Application Documentation and Deployment

Maintenance plays a key role in software engineering. It consumes the greatest proportion of expenses in the software life cycle. Although tools that support program comprehension on source code and component level are of great help, adequate documentation is the most obvious and effective way to support this comprehension process. Software documentation is a necessity to enable maintenance, and increasingly attention is being paid to it in practice.
The application documentation consists of the following sections:

- Requirements
- Architecture

- Functionality
- Tests
- Used components

Documentation, configuration and released application build the deployment package. The release is documented in the software package definition, in which the identification of the system and its components is presented. The release is based on the formal inspection of the test results.
Deployment manages the evolution of the application after it has been developed. It manages the install process, updating, reconfiguration and adapting software. An intelligent deployment tool can be used to manage, package, and deploy the component-based application on a set of embedded devices to the embedded device.

VI. Summary

The presented component-based development process subsumes the steps needed to develop applications out of components and in special cases the development of component itself. It concentrates especially on the composition phase, where we invest in methodologies enabling to predict the quality of a composed application as well as the stability. With enrichment of the component model with non-functional properties we are investigating in prediction of the behavior of composed systems, while we first address the need of small embedded devices i.e. micro controller time and memory consumption. The next steps we will investigate in are the validation of composition against the executions constraints of single components and validation of the scheduling, generated in consideration of these constraints. Furthermore we will investigate in the problem of version management of components and look for pragmatic approaches guiding software developers how to program components, which are exchangeable and so, enable system evolution.

VII. Acknowledgement

The work presented in this paper is part of the research project PECOS [7] granted by the European Commission under the IST Program IST-1999-20398.

VIII. References

[1] Oscar Nierstrasz, Gabriela Arévalo, Stéphane Ducasse, Roel Wuyts, Andrew Black, Peter Müller, Christian Zeidler, Thomas Genssler and Reinier van den Born, *A Component Model for Field Devices*, Component Deployment 2002, Berlin, June 20-21

[2] Watts S. Humphrey, *Managing the Software Process*, Addison Wesley Longman, Inc, 1989.

[3] Ivar Jacobson, Grady Booch, James Rumbaugh, *Unified Software Development Process*, Addison-Wesley, Object Technology Series, ISBN: 0201571692

[4] Alan Cameron Wills and Desmond francis D'Souza, *Objects, components, and Frameworks with UML – The Catalysis Approach*,. Addison-Wesley, 1999.

[5] Barry W Boehm, Software Engineering Economics, Prentice Hall, 1981

[6] P.O. Müller, C. Stich, C. Zeidler, *Components @ Work: Component Technology for Embedded Systems*, 27th Euromicro Conference, Euromicro Workshop on Component-based Software Engineering, Warsaw, Poland, September 4th – 6th, 2001

[7] The PECOS consortium. Pervasive Component Systems. http://www.pecos-project.org. 2000-2002

[8] Andreas Stelter, Software Process Guidelines, ABB Automation Products, 2001