Deliverable D3.1

Requirements for the Composition Environment

1. Identification

Project Id:	IST-1999-20398 PECOS
Deliverable Id:	D3.1 Requirements for the composition environment
Date for delivery:	2001-10-12
Planned date for delivery:	2001-03-31
Classification	Public
WP(s) contributing to:	WP 3
Author(s):	Benedikt Schulz (FZI), Thomas Genssler (FZI), Alexander Christoph (FZI), Michael Winter (FZI)

1.1 Abstract

This document describes the requirements for the composition environment. Starting from the presentation and discussion of usage scenarios, a set of requirements, that the composition environment should meet, is derived.

1.2 Keywords

Composition environment, requirements, specification, graphical composition, composition rules, glue code generation, repository, configuration management, deployment

1.3 Version History

Ver	*Date*	*Editor(s)*	*Status & Notes*
0.5	01-04-07	Benedikt Schulz	First draft
0.8	01-05-23	Thomas Genssler	Most of the requirements filled out, some spell checking
0.9	01-06-07	Alexander Christoph	Some more requirements
1.0	01-06-12	Benedikt Schulz	Final draft
1.1	01-09-24	Michael Winter	Some rewriting and updating. New section on component composition in PECOS.
1.3	01-10-08	Michael Winter	Reflect some comments from partners
1.5	01-10-10	Benedikt Schulz	Reflect rest of comments from partners
1.51	01-10-11	Thomas Genssler	Internal review, minor changes, summary

1.4 Classification

The classification of this document is done according to the security / dissemination level categories stated in Annex I (page 35) of the PECOS contract:

Classification	*Dissemination level*
Public (PU)	Public
Restricted (PP)	Restricted to other programme participants (including the Commission Services)
Restricted (RE)	Restricted to a group specified by the consortium (including the Commission Services)
Confidential (CO)	Confidential, only for members of the consortium (including the Commission Services)

1.5 Disclaimer

The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Table of Contents

2. Introduction

Component-based software engineering is a discipline that aims at constructing software out of components. Components are considered to be reusable software artifacts realizing a certain functionality. This functionality can be accessed through the components' interface – usually without the need to know anything about the internals (e.g., design, implementation) of the components. This allows for the so-called black-box reuse of components that happens while composing several components into one application. Black-box composition is the key explanation for being able to build software-systems of better quality within a shorter period of time – the main goals of component-based software-engineering. [Szyp98][Mons00][SG96][Sieg96][D 2.2.8-5]

Reducing development time while increasing product quality are exactly the challenges that engineers in the domain of field devices are facing. This is why they want to move to a component-based development of their embedded software which is currently done by hand-writing monolithic applications in C or assembly in a time-consuming and error-prone way. Supporting this paradigm shift towards the component-based engineering of software for field devices by appropriate methods and tools is the overall goal of the PECOS project.

Obviously modeling components and composing components into composite components representing the embedded software are the core intellectual tasks of this new approach. Methods and tools to support these tasks are therefore crucial to the success of the project. Whereas deliverable [D 2.2.8-5] describes the *PECOS component model* (e.g., it says what a component is, what properties a component has, what kinds of relationships between components exist and how they are composed) and deliverable [D 2.2.5] describes the syntax of the component description language *CoCo* in which (composite) components can be expressed, the deliverable at hand describes the requirements towards the respective tool-support, the so-called *Composition Environment*.

Besides supporting component-modeling and -composition the Composition Environment is responsible for interfacing with other PECOS tools like code- and test-case-generators, rule engines, repositories or (cross-)compilers. This is why the Composition Environment can be seen as the PECOS integrated development environment (IDE) that is collecting and managing all relevant data and running tools using that data. This IDE approach makes it clear, that most of the requirements described in this document are not requirements on the IDE but on the tools which interface with it.

The document is organized as follows: Section 3 outlines the main purpose of the composition environment and gives an overview on the development tasks that it will support. In addition, the overall architecture of the composition environment is sketched.

Section 4 gives a short introduction into the basics of component composition in PECOS, which is one of the fundamental ideas of the whole project. This section should help to get a clearer understanding about component and application development in PECOS and serve as a basis for the discussion in the following section.

Section 5 presents in detail the development tasks, which are supposed to be supported by the composition environment. This discussion is subdivided into usage scenarios concerning the role of the composition environment during component development, component composition, as well as deployment and testing.

Section 6 presents a set of requirements for the composition environment. The first part of the section is dedicated to requirements stemming from the usage scenarios of section 5. In the second part, requirements from other deliverables are discussed.

3. Introduction to the Composition Environment

3.1 Purpose and Scope

The composition environment is in the heart of the PECOS tool-set. It's main task is to provide an effective development environment that supports the developer during several steps of the PECOS development process. This process is roughly described in [D 1.1] and will be further elaborated in [D 5.2].

Developing components in the sense of deriving useful component specifications and interfaces is beyond the scope of this document. The composition environment comes into play when the first analysis phase is done, that means that there is already an idea of what the components should be and how their interfaces should look like.

Thus, the composition environment supports the tasks of application construction, testing and deployment. Application construction deals basically with component specification, implementation and composition. Testing comprises test case generation, test case execution and debugging. Finally, deployment deals with installing a PECOS application on a PECOS device which is a prerequisite for testing.

3.2 Architecture

As it becomes clear above, the composition environment has to support a wide range of activities from component implementation to application deployment. In order to cover all these tasks, the composition environment interfaces to specialised PECOS tools, like the skeleton and glue code generator, the rule checker, the compiler and so on. Figure 1 shows this concept, which is typical for many development environments.

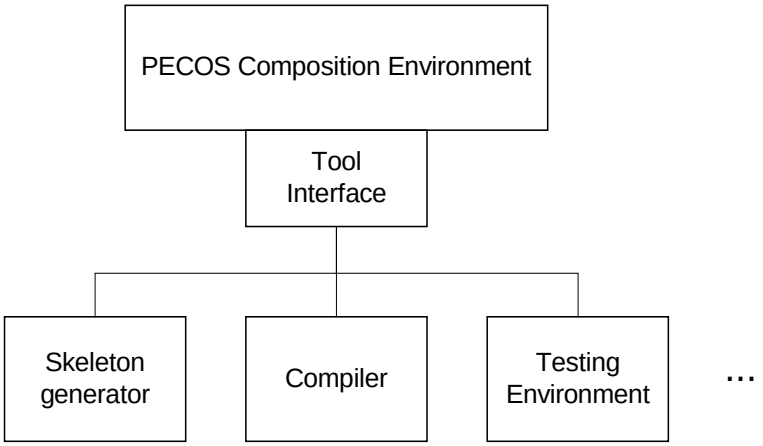

Figure 1: Global Architecture of the Composition Environment

4. Component Composition in PECOS

As pointed out, the main goal of the PECOS project is to construct applications for field devices from components. This means, that an application is built by configuring and composing components. Therefore, we will present the mechanism of composing components within the CoCo language [D 2.2.5] in some detail here, before taking a look at the usage scenarios. Building a PECOS application from components means creating a (composite) component that represents the application.

Let us explain how to build a composite component by giving a simple example. Assume, we have an `Adder` component, which can calculate the sum of two input float values. Input and output of components is modelled by so-called ports – places, where data flows into or out of a component. Such a component would have two in ports, which provide it with the values it should add. And it would have an out port, to which it writes the sum it calculates, whenever it is executed. In CoCo, this component would look like the following (see [D 2.2.5] for a detailed description of CoCo):

```
component Adder {
    in float value1;
    in float value2;
    out float result;
}
```

Now, we can build a component, which adds not only two `float` values, but three of them. Evidently, this can be done with two `Adder` components, the first one adding the first two values and the second one taking the result of the first `Adder` and adding the third value to it. Thus, we build the following

composite component, say `Adder3`, which consists of two instances of the above `Adder` component. In CoCo we get the following description:

```
component Adder3 {
    // here are the external ports of Adder3
    in float value1;
    in float value2;
    in float value3;
    out float result;

    // here are the two simple Adder instances, called adder1 and adder2
    Adder adder1;
    Adder adder2;

    // here are the wirings coming in
    connector c1 (value1, adder1.value1);
    connector c2 (value2, adder1.value2);
    connector c3 (value3, adder2.value2);
    // here are the internal wirings
    connector c4 (adder1.result, adder2.value1);
    // here are the wirings going out
    connector c5 (adder2.result, result);
}
```

In order to get a better idea of what is going on, figure 2 shows a simple graphical representation of the above CoCo specification.

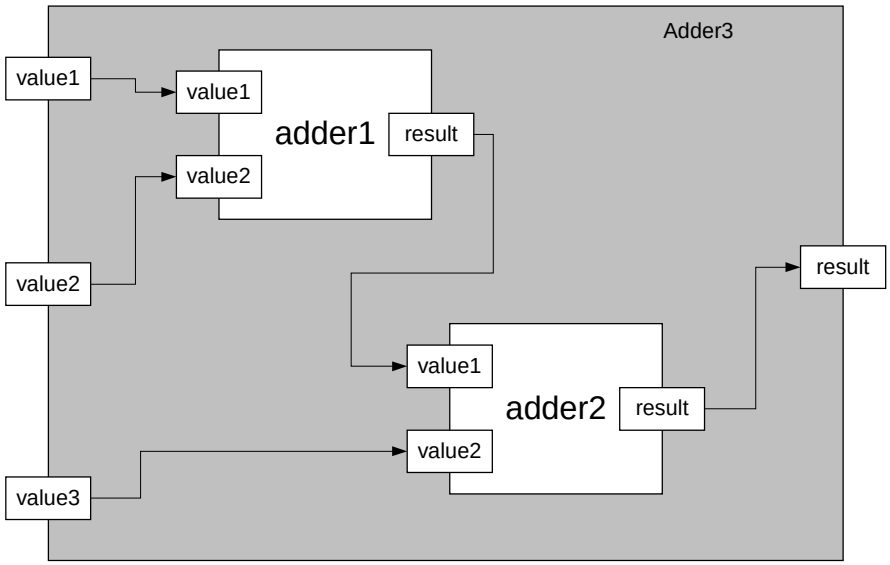

Figure 2: Graphical view of Component Adder3

The semantics of a connector (of a line in the diagram above) is the one of “a data-sharing relationship between ports” [D 2.2.8-5]. A connector represents a data variable, which is common to the connected ports. Writing data to a port, or reading data from a port means accessing the connector which connects the respective port to another one and thus means accessing the corresponding data variable.

In order to complete our `Adder3` example, we have to specify the order, in which the two `Adder` components have to be scheduled. Clearly, it should be `adder1` before `adder2`. Up to now it is not fixed, how the schedule is specified. This will be subject to further research. However it is clear that scheduling is crucial for the semantics of the application, which becomes clear if you consider `adder2` running before `adder1`.

In this section, we have presented what component composition means in PECOS. We have presented a simple example which illustrates, how more complex components (the Adder3 component) can be build from pre-build components (the Adder component) and how this is done in the CoCo language. The semantics of connectors as data dependencies has been sketched.

5. Tasks to be Supported by the Composition Environment

In this section we discuss the different scenarios, in which the composition environment plays a certain role. First, the role of the composition environment during component development is discussed. Then, the process of composing components in order to build more complex components and applications is presented. The section closes on deployment and testing scenarios.

5.1 Developing Leaf Components

As pointed out above, the first step in the PECOS development process which is supported by the composition environment concerns the implementation of leaf components, which means components that have no sub-components [D 2.2.8-5]. (Note: From “outside” a component leaf components and composite components can not be distinguished, because the fact whether a component has sub-components or not is hidden. [D 2.2.8-5] states, that “a field device is modelled as a component hierarchy, i.e., a tree of components with an active composite component as its root.” Obviously modelling can start top-down and bottom-up. We assume the latter and therefore describe the developing of leaf components first. Obviously developing leaf components is done the same way as developing composite components without the specification and connecting of sub-components.)

We refer to the PECOS development process in an idealised form, with strictly sequential subtasks. In a real development scenario, there would be certainly some kind of iteration. But this does not directly concern the requirements imposed on the composition environment. So, we can safely do this simplification here.

The following tasks have to be supported by the composition environment during this process:

- specifying the component within the CoCo language in terms of
    - ports
    - consistency rules
    - properties
- implementing the component
    - skeleton code generation: generating target code frame (C++ or Java classes) out of Coco description
    - filling out skeleton (done by application developer, cannot be generated!)
- compilation of component
- documenting the component
- storing the component together with its documentation in the repository

First, the developer specifies the component’s entities in the CoCo language. This consists of describing the data ports, consistency rules and properties of the component. We have already seen the use of ports in the example presented in section 4. Consistency Rules are used to express properties that make sense even for leaf components without composition. They will be described in detail in the respective deliverable [D 3.3], so for this document is sufficient to state that they express rules on the existence and on values of component properties. For a discussion on component

properties refer to [D 2.2.5] and [D 2.2.8-5]. For the task of specifying components, the developer, especially in the domain we are considering[1], normally uses a text editor.

The implementation of the component is done in two steps: First, the Coco-description of the component is used to generate a target code frame (that is, C++ or Java classes). This is referenced to as *skeleton code generation*. Second, the skeleton is filled out by the application developer by adding the concrete actions (e.g., reading values, changing values, storing values) the component is to perform.

The next step is to compile the implemented component in order to find implementation errors, local to the component. Compilation is triggered from within the composition environment. Errors occurring during compilation are returned to the composition environment, where they can be viewed and browsed by the developer. (Note: The compiling and testing of one single component may require test-bed and driver-generation. Although not all checks can be already done on parts of the system, a compiled and "running" single component is important to be able to determine and verify non-functional requirements like execution time and memory consumption. Deliverable [D 2.2.7] will deal with these issues in more detail. Most probably, leaf components will not be tested on the target device but in a testing environment.)

Then, the developer has to document the component he has just created. For this task, one can use a documentation tool, which is also interfaced by the composition environment.

Finally, the newly created component is stored in the repository together with its implementation and documentation. (Note: There was a decision not to spend any effort on repositories, because a "persistent repository and an associated query language become relevant only when there are a lot of components and when these components need to be accessed by several developers." [D 6.3] This is, however not the case at this time.)

5.2 Developing Composite Components

Now, we take a look at the development of composite components out of already built ones. Of course, in a real development scenario, this step would be mixed up with the above one (see comments in the previous subsection), but for simplicity we assume here that all required (leaf) components have already been built.

The following tasks have to be supported by the composition environment during this process:

- browsing the repository in order to find components to be used to build the composite component
- inspecting and setting property values of these components
- specifying the composite component in terms of
    - wiring of its sub-components by connectors
    - ports
    - consistency and composition rules
    - properties
- implementing the component
    - skeleton code generation: generating target code frame (C++ or Java classes) out of Coco description
    - filling out skeleton (done by application developer, cannot be generated!)
- compiling the component

[1] Software for field devices is usually developed using normal text-based development IDEs (like Microsoft Visual Studio) or even with normal text editors like Emacs and command-line based tools like gcc.

- documenting the component
- storing the component together with its documentation in the repository

First, the knowledge about the composite component has to be used to find already existing components in the repository. Once the necessary components are found, they need to be loaded into the composition environment for further investigation and – if necessary – adaptation.

Then, the components can be composed together in order to build the composite component. This means that the developer decides, which component ports have to be connected together, as we have described in the example of section 4. In addition, scheduling information has to be provided.

The rest of the process is similar to the simple case and is therefore not repeated here. (Note: Composition rules express rules on the existence and values of certain properties: the sub-components. Details are left to [D 3.3])

5.3 Deployment and Testing

The tasks supported by the composition environment during deployment and testing are the following:

- consistency rule checking
- test case generation and execution
- downloading the application to the device
- debugging the application on the device

Rule checking is a central idea in PECOS. The goal is to construct applications, which are correct by construction with respect to certain design rules. As pointed out earlier, rule checking will be the topic of [D 3.3].

Testing is another technique to eliminate errors from an application before deployment. The composition environment has to provide a tool interface for this purpose.

There should certainly be support for installing a ready PECOS application on the field device. And there should also be the possibility to debug the application right on the filed device from within the composition environment.

All these tasks rely heavily on the PECOS execution environment, which will be in focus of work package 4.

6. Requirements

In this section all the requirements that can be derived from the usage scenarios are presented. In addition, we include requirements stemming from [D 1.1] and comment on them. Any requirement is attributed with a priority (given in brackets). Priority levels are *high*, *medium* and *low*.

6.1 Requirements coming from the usage scenarios

6.1.1 General

R01 Textual Representation of Model Entities (high)

The model entities must have a textual representation (plain ASCII) in order to incorporate existing development tools for standard tasks (e.g., standard editors for editing component specifications, etc.). The syntax and semantics of this textual representation is specified by the CoCo language [D 2.2.5].

R02 Graphical Representation of Component and Project Structure (high)

The composition environment must provide support for a GUI like representation for the component and project structure of composite components. We suggest a tree view or similar representations known from state-of-the art development tools. This view should provide folding capabilities i.e., collapsing/expanding component elements in order to support quick browsing.

6.1.2 Component Specification

R03 Editing CoCo component specification (high)

The composition environment must provide support for editing component specifications in CoCo during component specification. Syntax highlighting should be provided.

6.1.3 Component Adaptation

R04 Modifying component properties for adaptation (high)

Adaptation of PECOS components means setting or changing the component's properties. Thus, the composition environment must provide support for the modification of component properties during component adaptation.

6.1.4 Component Composition

R05 Adding Component Instances (high)

The composition environment must provide support for adding component instances to a composite component. These instances may come from the local project or from the repository.

R06 Textual wiring components together on model level (high)

The composition environment must provide support for wiring ports of the components within a composite component on a textual basis.

6.1.5 Code Generation

R07 Skeleton generation (high)

The composition environment must provide a tool interface for skeleton generators.

R08 Scheduler generation (high)

The composition environment must provide a tool interface for scheduler generators.

6.1.6 Component Implementation

R09 Handling of target language code (medium)

The composition environment must provide support for adding and modifying target language code for the different supported target languages to a component. The actual modification of target language code must be supported by providing an internal text editor or by interfacing with an external one. Once added, the target language code must also be stored within the repository. (Note: Since the PECOS approach is model-based, only the hand-written code can be changed. Generated target code frames (skeletons) should not be changeable.)

6.1.7 Compilation

R10 Compilation of components (high)

The composition environment must be able to interface to compilers for different languages.

6.1.8 Documentation

R11 Generating documentation (low)

The composition environment must provide support for generating javadoc like documentation out of component specifications.

R12 Documenting a component (low)

The composition environment must provide support for editing additional component documentation. This documentation can either be plain ASCII text or formatted HTML.

6.1.9 Repository Access

R13 Interface to repository for querying existing components (low)

The repository is the basic storage for already defined components. Beside simple browsing facilities, the repository should also support search queries, which specify the component properties.

R14 Loading components in from the repository (low)

The composition environment must be able to load or export component specifications from the repository into the project or working directory. The components must be selectable by simple browsing the repository, or by performing queries, which select components by specifying their properties.

R15 Storing version/update of component back to repository (low)

The repository must be able to store different versions of a component specification. Also updates of existing versions must be possible.

R16 Storing composed components in the repository (low)

New components, which can also be compositions of existing components, can be stored back into the repository to support later usage of these components. Therefor the repository supports different versions and updates of existing versions (see R12).

6.1.10 Design Rules

R17 System composition rule checking (high)

The composition environment must interface to a tool, which can be used to do system checking using composition rules. Composition rules will be described in [D 3.3].

6.1.11 Conformity Checks

R18 Model conformity checks (high)

The composition environment has to provide an interface for a conformity checking tool.

This tool has to support for checking the conformance of a component specification with respect to the PECOS model. This includes syntactical checks as well as semantic checks on the level of the underlying ADL, such as “all used identifiers have been declared”, type checking (i.e., signatures of connected ports match), etc.

6.1.12 Testing

R19 Interfacing to a testing tool (high)

The composition environment must interface to a testing tool, which can be used to generate test cases and execute these test cases.

6.1.13 Deployment

R20 Interfacing to a deployment tool (high)

The composition environment must interface to a deployment tool, which enables the developer to download his application to the PECOS device.

6.2 Requirements coming from [D 1.1]

In this subsection we discuss the requirements stemming from [D 1.1] concerning the composition environment.

6.2.1 General

FR55 Customisation GUI (high)

The composer has to provide a user interface for customisation of instances of components. Herein e.g., properties can be set.

see R04.

FR56 Engineering Data Export (high)

An export interface has to be provided for the engineering data generated in FR57.

See discussion of FR57.

FR57 Engineering Support (high)

Configuration data for the infrastructure have to be automatically generated from the block and parameter configuration. These can then be used to create configuration software for this particular field device and application (→ FDT, DTM etc.)

It is unclear, what block and parameter configuration means. May be components and properties? Anyway, we don’t see this requirement being essential to the project.

6.2.2 Component Composition

FR58 Graphical Composition (low)

Components are represented as blocks. These blocks can be selected from a menu and placed on a sheet (instantiation). A sheet represents an architecture template and can also be selected from a menu. Blocks have stubs by which they can be connected to other blocks. During the composition the design is validated based on the composition rules which are part of the template. The stubs represent the ports and lines between the blocks visualise connectors. The connectors are the "glue code" between the components.

We don't see the graphical composition as a major requirement. If possible, the composition environment will interface to existing graphical editors, even better would be using graphical editors from OTI if available. The major innovation in PECOS is not the graphical support of the composition but the correct-by-construction software composition as realised by the model-based approach and the different composition rules. We therefore strongly suggest to release this requirement. Note: This suggestion was already confirmed [D 6.3].

FR59 Composer API (high)

The composition environment has to provide an interface to programmatically access the composer functionality.

The "composer functionality" is provided on several levels: Specifying composite components with Coco, generating skeletons and schedulers and hand-writing component code in a target language. All that should be possible without the IDE (=composition environment).

6.2.3 Code Generation

FR60 Glue Code Editor (low)

An editor for glue code has to be provided. Code that is used to fill the gap between components. This corresponds to designing new classes of connectors.

This requirement should get a very low priority because there is only one connector type in the model and thus there is no need to editing glue code.

6.2.4 Repository Access

FR61 Repository Integration (low)

Search inquiries on the component repository can be executed from within the composition environment. The search results are presented in a list or menu. Component updates are also possible. Tested applications and components can be checked-in into the repository.

The facilities of the composition environment concerning repository integration depend heavily on the facilities of the repository and the complexity of its interface. The composition environment will make as much use of these facilities as possible and sensible concerning the overall project goals.

6.2.5 Design Rules

FR62 Composition Rule Designer (high)

An editor for composition rules has to be provided. These rules are stored as part of architecture templates which are used as starting point for application development.

Composition rules can be classified under two categories: *computation rules* and *validation rules*.

Computation rules are used to derive non-functional properties of composites from their underlying components.

Validation rules are used to validate a composition based on its non-functional properties and constraints.

It is not yet clear, how rules will be used for in PECOS. This will be described in detail in [D 3.3].

FR63 Validation (high)

The composition environment has to provide functionality to validate the design at design-time. This validation is based on constraints and composition rules.

Constraints could be the overall execution time, the power consumption etc. These values are either properties of components or calculated values on composite components (composition of components). The validation can either be triggered automatically whenever blocks are placed on the sheet and connected or manually by the user.

This kind of validation is a core innovation of the PECOS project. Therefore the composition environment should provide as much support as possible. Details of what can be checked and how are, however, still to be elaborated and therefore left to [D 3.3].

6.2.6 Testing

FR64 Test Case Generation (low)

The development environment has to provide an editor to define test cases. These test cases can then be generated partly automated.

Test case generation is considered to be done by a specialised tool. Thus it is not part of the composition environment directly. The composition environment only has to provide a tool interface, which supports for plugging such a tool.

FR65 Test Case Execution (low)

A test framework has to be provided that allows for test suite-based automated testing of applications.

Test case generation is considered to be done by a specialised tool. Thus it is not part of the composition environment directly. The composition environment only has to provide a tool interface, which supports for plugging such a tool.

6.2.7 Deployment

FR66 Automated Deployment (low)

The deployment has to be (more or less) a one-step deployment process (for the user) and not an error-prone sequence of tasks.

Application updates are directly validated before downloading to the target platform. If an update has not proven to be valid the changes will not go in operation.

Test case generation is considered to be done by a specialised tool. Thus it is not part of the composition environment directly. The composition environment only has to provide a tool interface, which supports for plugging such a tool.

FR67 RTOS Configuration (low)

Configuration parameters for the run-time environment have to be derived. These are used during application building and deployment.

This is more a requirement to the ultra-light component environment rather than the composition environment.

FR68 Selection of Target Platform (low)

The target platform to deploy the application on has to be selected. The development platform connects then to the target platform and further actions like e.g., download can be executed.

This is more a requirement to the ultra-light component environment rather than the composition environment.

FR69 Concurrent Configuration and Development (low)

The composition environment has to support concurrent development. The granularity for access has to be on application level so that only one user can manipulate an application at a time.

Concurrent development is a question of transaction management or entity locking and releasing. Although we see that this requirement is important for a production tool, it's beyond the scope of the prototypical implementation of the composition environment provided during the scope of PECOS.

7. Summary

In this document we first gave a short introduction into the composition environment and its overall architecture. After discussing the process of component composition we presented a list of use cases, the composition environment has to support. Starting from these use cases, concrete requirements on the composition environment were derived and discussed. We also discussed requirements which originated from [D 1.1] and related them to the use case discussed earlier.

The most important shift in the requirements compared to [D 1.1] and the original approach for the composition environment was that we decided to assign a low priority to both a graphical composition tool, where components are composed in a graphical way, i.e., by drawing lines an boxes and the interface to the repository.

8. Bibliography

[D 1.1] Field Device Requirements

[D 2.2.5] Description of the PECOS Component Model and the COCO language

[D 2.2.6] Verifying scheduling, timing and memory consumption of components.

[D 2.2.8-5] Field-Device Component Model-V (previously D 2.2.1 and D 2.2.2)

[D 2.2.7] The Component Instrumentation Environment.

[D 3.3] Composition rules for case study

[D 4.1] C++/C-based ultra-light component environment

[D 4.2] Java-based ultra-light component environment

[D 4.4] Remote Embedded Component Debugger

[D 5.2] Manual for the cookbook

[D 6.3] Refined and updated project plan

[Mons00] R. Monson-Haefel, Enterprise Java Beans, O'Reilly, 2000.

[SG96] M. Shaw and D. Garlan. Software Architecture – Perspectives on an Emerging Discipline. Prentice Hall, 1996.

[Sieg96] J. Siegel, Corba: Fundamentals and Programming, John Wiley & Sons, 1996

[Szyp98] C. A. Szyperski. Component Software. Addison-Wesley, 1998.