Deliverable D 3.3

Report on composition rules

1 Identification

Project Id:	IST-1999-20398 PECOS
Deliverable Id:	D 3.3-3 (Living document)
Date for delivery:	2001-12-31
Planned date for delivery:	2001-12-31
Classification	Public
WP(s) contributing to:	WP 2 (“Meta model”), WP 3 (“Composition Environment”)
Author(s):	Christoph (FZI), Genssler (FZI)

1.1 Abstract

The PECOS component model together with its composition language provides a powerful way to specify component software for embedded systems. This document defines and classifies composition rules and the PECOS rule checking technology together with an example.

Keywords

PECOS, CoCo, composition rules, Architectural Description Language, Architecture

1.2 Version history

Ver	*Date*	*Editor(s)*	*Status & Notes*
3.1	24.09.02	A. Christoph	Example rule
3.1	24.09.02	T. Genssler	Minor corrections
3.0	23.09.02	A. Christoph	Description of rule checker added, document updated
2.2	26.02.02	A. Christoph	Updated literature list; added example.
2.1	06.02.02	A. Christoph	Included application specific rule examples
2.0	29.01.02	A. Christoph	Updated document structure
1.0	10.12.01	A. Christoph	First draft

1.3 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

1.4 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.

2 Table of Contents

3 Introduction

Requirements for embedded devices and their software are much higher than for regular desktop programs. Embedded devices are used in safety-critical environments and therefore have to be designed carefully. Syntactic language rules alone are not sufficient to guarantee correct programs that fulfill real-time and resource requirements. Certain design-principles have to be followed to construct software that meets these requirements. Therefore the CoCo language [COCO01] allows the developer to specify functional and non-functional properties of their programs and components. The Pecos modeling enviroment provides a rule checking facility to allow reasoning about these functional and non-functional properties as well as the structural properties of CoCo programs.

This report explains the approach chosen to support reasoning. It classifies rules which can be checked within the Pecos environment using the Pecos rule checker. The report briefly sketches the general approach to reasoning about program properties. It also describes the Pecos rule checker, its Prolog-based implementation as well as its usage. Examples are provided to show how the generated fact base looks like together with some example rules.

This document does not give, however, a complete overview on composition rules important in the field of Pecos. The reason for this is that Pecos technology is currently in the process of being put into practice. This means that there are only very view practical experiences on how good programming style in CoCo should look like. We therefore focus on introducing the essentials for specifying future composition rules instead of providing a set of pre-defined rules.

4 Composition rules

In our understanding, composition rules serve to formalise expert know-how about good programming style on the level of composition. This means that they describe how entities of an application (such as components, objects, classes, methods) shall relate to each other in a correct way[1]. Examples of composition rules are design heuristics (e.g., [Riel96]), design patterns (e.g., [GHJV95]) and recipes. We distinguish language-specific rules, paradigm-specific rules and rules imposed by a certain application or application domain.

Language-specific rules

We distinguish syntactic and semantic rules. Syntactic rules define the well-formedness of a composition or of a part of it. Semantic rules define typing, scoping, definition/use of composition elements. Rules of this type are usually checked by providing some context-free grammar describing the valid structure of programmes along with semantic rules (described by context-sensitive grammars). Language-specific rules are usually checked by the language compiler itself and are thus not in the scope of the composition rule checker.

Paradigm-specific design rules

Paradigm-specific rules describe design patterns and design heuristics which are characteristic for a certain programming paradigm. These rules are usually not checked by a language compiler. Examples of paradigm specific rules in the object-oriented paradigm are: “A class should never have knowledge about its sub-classes!” or “A class should capture one and only one key abstraction”([Riel96]).

Domain- and application-specific rules

These rules define heuristics or patterns emerging from a particular application domain or a particular application. Examples can be

- In a layered architecture, there must not be dependencies from lower layers to upper layers.
- An EEPROM controller shall always be connected to the rest of the device via a buffer component.

[1] Note that syntactic and semantic rules at the level of statements (such as evaluation order of expressions and the like) are not in the scope of our understanding of the term ‘composition rules’.

The Pecos rule checker aims at checking the compliance of an application to the latter two kinds of rules.

5 The general approach

This section describes the general approach to checking composition rules which we have chosen for Pecos. This approach is based on the use of first-order predicate logic. Although this somewhat limits the expressive power for specifying rules[2], this approach has proven its applicability and practicability in other situations [CIU99], [CIU01].

The first step is to collect knowledge about the system. The information collected comprises entities defined by the system according to the entity types given by the meta model of the language under consideration (CoCo in our case), such as components, ports, connectors, etc. as well as relations between these entities, such as `definesPort`, `hasPropertySet`, etc. Details of the Pecos meta-model can be found in [D-2.2.8] [D-2.2.5].

This knowledge is represented as atomic predicates or facts:

fact(o).

or

fact(O).

fact... *the entity or relation type*
o... *the object*
O... *a set of objects specifying the objects for which a particular relationship is valid*

These set of all facts together form the **fact base**.

Now inference rules have to be specified to infer new (implicit) knowledge about a system. These inference rules have the form of:

$goal(X) \leftarrow pred_1((X_1 \subseteq X) \cup V_1), pred_2((X_2 \subseteq X) \cup V_2), ..., pred_n((X_n \subseteq X) \cup V_n).$

X... *a set of variables and/or constants*
$pred_i$... *facts or other (composite) predicates*
X_i... *the sub-set of variables/objects of goal(X)*
V_n... *additional variables/objects (can also contain bindings of variables of X*

The meaning of such an inference rule is that when all $pred_i$ hold for X than *goal(X)* also holds for X. With other words: from $pred_1$ *and* $pred_2$ *and ... and* $pred_n$ follows *goal*.

If X contains variables, each defined object in the fact base in each possible combination is bound to these variables. A valid binding of objects to variables is one for which $pred_1$ *and* $pred_2$ *and ... and* $pred_n$ hold. If X contains variables and there are several valid bindings for these variable (so-called modells) then goal(X) yields multiple solutions (one for each valid binding). The predicate *goal(X)* can again be used to construct more sophisticated predicates.

Composition rules are either specified using anti-patterns (in order to identify violations of a rules or to make sure that certain anti-patterns to not exist, negative serach) or in a 'positive' form (in order to make sure that certain patterns exist, positive search).

The actual querying is performed using so-called goal clauses or queries. These queries have the form:

$goal_1((X_1 \subseteq X) \cup V_1), goal_2((X_2 \subseteq X) \cup V_2), ..., goal_n((X_n \subseteq X) \cup V_n).$

$goal_i$... *a certain predicate for which all possible models (all valid binding of variables in* $((X_i \subseteq X) \cup V_i)$ *to objects in the fact base) are computed*

X_i ... *set of variables used for* $goal_i$

[2] Sometimes, temporal dependencies might be useful. However, employing modal logics is usual expensive and often does not scale. It shows on the other hand that most theoretical problems (if the can represented in an axiomatic way at all) can be represented using first-order predicate logic.

V_1... *set of addition variables/objects used for* $goal_i$

X... solution variables. These variables are bound to objects which form a solution of the given query

Each solution *X* of a query represents an occurence (a set of objects) of the pattern (or anti-pattern respectively) specified by the conjunction of the clauses $goal_1(X), goal_2(X), ..., goal_n(X)$.

In a final step, these occurences have to be analyzed and potential violations of patterns have to be fixed.

In the following we give an example of how such rules look like and how the compliance to, or violation of such rules can be checked using queries.

Two example rules are used to show the approach in practice. The first example rule tests, if there is at least one active component in the project. The second example rule tests, if the given component is active and has a task definition.

```
hasActiveComponent:-component(_, 'active').
isMainComponent(X):-component(X, 'active'), task(_, X, _, _, _).
```

For a detailed description of the generated predicates, see chapter **Error! Reference source not found.**.

6 Implementation

The PECOS rule checker is implemented using Prolog. Prolog provides the expressivness of first-order predicate logic in the form of Horn clauses. Horn[3] clauses have the form of:

$$P \leftarrow Q_1, Q_2, ..., Q_n.$$

with the right side of the clause being the conjunction of all Q_i and each Q_1 contains at most one positive literal. Horn clauses can also be represented as a conjunction of implications:

$$P \leftarrow ((p_1 \rightarrow q_1) \wedge (p_2 \rightarrow q_2) \wedge ... \wedge (p_n \rightarrow q_n)).$$

The meaning of this is that *P* follows iff for all p_i follows q_i. For a more detailed introduction to logic programming with Prolog see for example [SWI].

The rule checker consists of three parts: the Prolog plugin, the fact generator, and the Prolog engine. The first is the user interface to the rule checker. The fact generator serves to extract knowledge from a CoCo project. These facts along with rules are loaded into the Prolog engine which does the actual rule checking. All parts of the rule checker are fully integrated with Eclipse.

6.1 The Prolog-Plugin

The Prolog-Plugin is the main entry point for rule checking purposes in the Pecos modelling environment. It offers a user interface to trigger the fact generation, load already defined rules and issue queries about the current project. Figure 1 shows a screen-shot of the Prolog-Plugin.

[3] Named after the logician Alfred Horn.

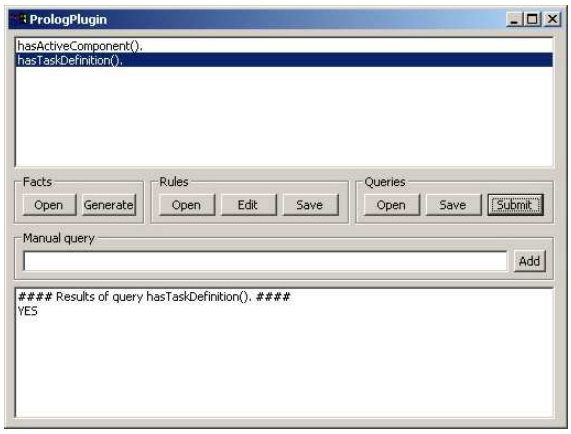

Figure 1, User interface of the Prolog plug-in

6.2 The Prolog fact generator

The fact generator generates a set of Prolog facts describing a Pecos project. The generator uses all information available to describe all aspects of the project. The code example shows a fragment of a fact base.

```
component('DigitalDisplay', 'passive').
port('DigitalDisplay', 'optional', 'input', 'long', 'time_in_msecs').
port('DigitalDisplay', 'optional', 'input', 'bool', 'can_draw').
```

See chapter **Error! Reference source not found.** for further details on the generated facts.

6.3 The Prolog-Engine

The Prolog engine is responsible for executing (solving) queries against the fact base of the current project. It consults the generated fact base and the composition rules. It then allows the user to issue queries about the current project and returns the found answers. We use SWI-Prolog [SWI] as an open source prolog engine.

6.4 Usage

The Prolog-Plugin is the user interface to the PECOS rule checker. The user opens the dialog window by clicking on a toolbar button. The window consitis of four parts: a list of queries, a button section, a textfield for defining new queries and a textarea for displaying the results of the Prolog engine.

When the user wants to verify, that a PECOS application conforms to a set of rules, he has to select the project in the resource navigator window of the eclipse IDE. He then starts the dialog of the Prolog plug-in by clicking on the plug-in's toolbar button.

Generate facts

In order to verify the project, Prolog facts have to be generated, that represent all entities and relationships in the project. The fact generator is started by pressing the *Generate* button in the button section of the dialog. If the user selected a project in the navigator, facts for this project are generated. Otherwise, the user is asked to select one of the available projects. The generated facts are automatically consulted by the Prolog engine.

Load or edit rules

After generating facts, the user can now load and edit a set of rules. By pressing the *Load* button, the user is asked to select a rule file. With *Edit*, the user can edit these rules and/or enter new rules. *Save* saves the rule set and lets the Prolog engine reconsult the new rules.

Issuing queries

Queries can be loaded and saved just like rules. After loading queries, they are listed in the query list in the top section of the dialog. A query can be selected and executed by pressing *Submit*. New queries can be entered in the list through the query textfield in the middle of the dialog. The results of a query are displayed in the lower textarea of the dialog.

7 Prolog facts and rules

This chapter describes the generated Prolog fact-base, together with the used predicates. Section 7.1 explains the predicates in detail, while section 7.2 shows an example fact-base generated from an example PECOS project.

7.1 Prolog predicates

This section describes the generated Prolog predicates in detail. See [D-2.2.8]] for details about the Pecos meta model.

7.1.1 Components

Predicate	Description
`component(name, type)`	*name* is a component with the given *type*; *type* can be 'active', 'passive' or 'event'
`implements(name, list_of_names)`	component *name* implements the listed abstract components
`port(cname, modifier, direction, datatype, name)`	the port *name* is defined with the given *modifier* ('optional', 'mandatory'), *datatype* and *direction* ('input', 'output', 'inout'); the owning component is *cname*
`instance(cname, type, name, role)`	the instance *name* has the given *type*, binds the given *role* and is defined in *cname*
`connector(cname, name)`	connector *name* is defined in component *cname*
`connect(name, port)`	*port* is connected to connector *name*
`abstractcomponent(name)`	*name* is an abstract component
`extends(name, list_of_names)`	abstract component *name* extends the listed abstract components
`role(cname, type, name)`	defines the role *name* with the given *type* in the abstract component *cname*

7.1.2 Datatypes

Predicate	Description
`datatype(name, type)`	defines *name* as datatype; *type* can be 'extension' or 'definition'
`builtintype(name)`	defines *name* as built-in type
`basetype(name, btname)`	datatype *btname* is the base type of *name*
`field(dtname, type, name)`	field *name* with the given *type* is member of datatype *dtname*

7.1.3 Property sets

Predicate	Description
`propertyset(name)`	defines *name* as property set
`property(pname, ...)`	defines property *name* with *modifier* ('optional' or 'mandatory') and

`modifier, name, value)`	*value*; parent is *pname*
`propertysetref(pname, name)`	defines a reference from *pname* to property set *name*

7.1.4 Tasks

Predicate	Description
`task(name, instance, base, cycletime, priority)`	defines task *name* of *instance*, inheriting from *base* with *cycletime* and *priority*
`job(pname, name)`	defines job *name* of task *pname*
`jobScheduled(pname, name, time)`	job *name* of task *pname* is scheduled at *time*
`activity(name, instance, type)`	defines activity *type* (’execute’ or ’syncronize’) on *instance* of job *name*

7.2 Example fact-base

This chapter shows an example fact-base, generated from the example application, described in chapter 2 of the PECOS handbook [HB02]. The different sections of the file are commented. Figure 2 shows the structure of the example application.

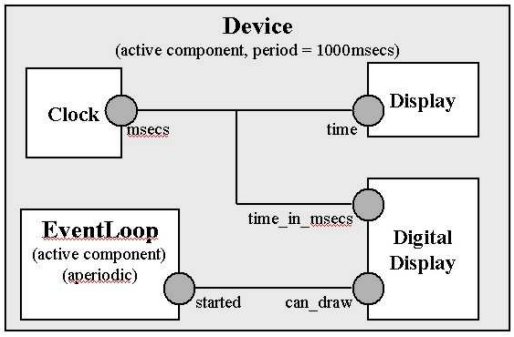

Figure 2, Example application

Built-in datatypes of the CoCo language.

```
% List of built-in types.
datatype('bool', '').
builtintype('bool').
datatype('byte', '').
builtintype('byte').
datatype('char', '').
builtintype('char').
datatype('double', '').
builtintype('double').
datatype('float', '').
builtintype('float').
datatype('int', '').
builtintype('int').
datatype('long', '').
builtintype('long').
datatype('short', '').
builtintype('short').
datatype('void', '').
builtintype('void').
```

Component Clock with output port msecs.

```
component('Clock', 'passive').
property('Clock', 'optional', 'wcMemUsage', '32').
% port msecs [2-2]
port('Clock', 'optional', 'output', 'long', 'msecs').
```

Component DigitalDisplay.

```
% component DigitalDisplay [1-4]
component('DigitalDisplay', 'passive').
property('DigitalDisplay', 'optional', 'wcMemUsage', '128').
% port time_in_msecs [2-2]
port('DigitalDisplay', 'optional', 'input', 'long', 'time_in_msecs').
% port can_draw [3-3]
port('DigitalDisplay', 'optional', 'input', 'bool', 'can_draw').
```

Component Display.

```
% component Display [1-3]
component('Display', 'passive').
property('Display', 'optional', 'wcMemUsage', '24').
% port time [2-2]
port('Display', 'optional', 'input', 'long', 'time').
```

Component EventLoop.

```
% component EventLoop [1-3]
component('EventLoop', 'active').
property('EventLoop', 'optional', 'wcMemUsage', '17').
% port started [2-2]
port('EventLoop', 'optional', 'output', 'bool', 'started').
```

Component Device with instances and connectors forms the whole application.

```
% component Device [1-10]
component('Device', 'active').
property('Device', 'optional', 'wcMemUsage', '192').
property('Device', 'optional', 'memAvail', '3000').
% instance clock [2-2]
instance('Device', 'Clock', 'clock', '').
% instance display [3-3]
instance('Device', 'Display', 'display', '').
% instance digitalDisplay [4-4]
instance('Device', 'DigitalDisplay', 'digitalDisplay', '').
% instance eventLoop [5-5]
instance('Device', 'EventLoop', 'eventLoop', '').
% - connectors
connector('Device', 'time').
connect('time', 'clock.msecs', []).
connect('time', 'display.time', []).
connect('time', 'digitalDisplay.time_in_msecs', []).
connector('Device', 'eventLoop_started').
connect('eventLoop_started', 'eventLoop.started', []).
connect('eventLoop_started', 'digitalDisplay.can_draw', []).
```

Task sched of component Device and activities for enclosed instances.

```
% task sched [12-19]
task('sched', 'Device', '', '1000', '10').
jobScheduled('sched', 'jobAt_0', '0').
activity('jobAt_0', 'eventLoop', 'synchronize').
activity('jobAt_0', 'clock', 'execute').
activity('jobAt_0', 'display', 'execute').
activity('jobAt_0', 'digitalDisplay', 'execute').
% task eventTask [21-25]
task('eventTask', 'Device.eventLoop', '', '0', '5').
jobScheduled('eventTask', 'jobAt_0', '0').
activity('jobAt_0', '', 'execute').
```

7.3 Example Rules

This example presents a set of rules, to verify the correctness of a component composition regarding the memory consumption. Therefore every component defines a property wcMemUsage, which defines its maximum memory footprint. The enclosing active top-level component, which models the root of the PECOS application additional specifies the property memAvail, which defines the maximum availabe space in memory. A component composition is considered correct, if the sum of all memory amounts of all child components is less or equal the maximum of available memory space.

The first rule extracts the `memAvail` property of the given top-level component and compares it to the sum of the requested memory of all child components.

```
checkMemConsumption(D):-
        component(D,'active'),
        property(D, _, 'memAvail', M1),
        findall(I, instance(D,I,_,_), InstList),
        sumMemConsumption(InstList, R),
        string_to_int(M1, M),
        R=<M.
```

The second rule sums the values of all `wcMemUsage` properties of a given set of components. The rule is called on the list of child components of the device component.

```
sumMemConsumption([],R):-R=0.
sumMemConsumption([H|T], R):-
        sumMemConsumption(T, R1),
        component(H,_),
        property(H,_,'wcMemUsage',U),
        string_to_int(U,I),
        plus(R1,I,R).
```

The rules are defined by the user in a seperate editing window, or loaded from an external file. The user can then ask the system to verify the memory consumption of the current application. He issues the following command in the manual query text field.

```
checkMemConsumption('Device').
```

The systems responds with "Yes" or "No", depending on the values of the component properties.

8 Conclusions

This report defined and classified composition rules from the perspective of CBSE. It was explained, that the scope of this report is beyond syntacitic and semantic language definitions, that can be checked by standard language tools, like parsers. Paradigm-, as well as domain- and application-specific composition rules are required to build reliable and secure software for critical embedded applications. The report described the PECOS approach to support reasoning on these rule classes, its theoretical background and ist implementation. Examples were provided, that explain generated facts as well as example rules.

9 References

[COCO01] T. Genssler, *The CoCo Grammar*, Code Document, 2001

[D-2.2.5] A. Christoph, T. Genssler. *Description of the CoCo architectural description language and composition environment*, Pecos-Deliverable 2.2.5

[D-2.2.8] R. Wuyts et al. *Specification of the model for embedded components*, Pecos-Deliverable 2.2.8

[GZ01] T. Genssler, C. Zeidler. *Rule-driven component composition*, Proceedings of the 4th ICSE Workshop on Component-Based Software Engineering, Component Certification and System Prediction, 2001.

[PRO02] M. Winter. *Pecos Process Summary*, Achievement Report, 2002.

[HB02] T. Genssler, A. Christoph, B. Schulz, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arevalo, P. Liang, P. Müller, C. Stich, C. Zeidler, A. Stelter, B. Schönhage, R. v.d. Born. *PECOS in a nutshell.*

[LCTES] T. Genssler, A. Christoph, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arevalo, P. Müller, C. Stich, B. Schönhage. *Components for Embedded Software – The PECOS Approach.*

[SWI] Homepage of SWI-Prolog. http://www.swi-prolog.org

[Riel96] A.J. Riel. *Object-oriented Design Heuristic*s. Addison-Wesley, 1996.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[CIU99] O. Ciupke. *Automatic Detection of Design Problems in Object-Oriented Reengineering*. In D. Firesmith , R. Riehle, G. Pour, B. Meyer (editors): Technology of Object-Oriented Languages and Systems - TOOLS 30, 18–32. IEEE Computer Society, 1999.

[CIU01] O. Ciupke. *Problemidentifikation in objektorientierten Softwarestrukturen*. Phd Dissertation, Institut für Programmstrukturen und Datenorganisation, University of Karlsruhe, FZI Karlsruhe, 2001.