Components for Embedded Software

The PECOS Approach

Thomas Genßler
Forschungszentrum Informatik (FZI), Germany,
http://www.fzi.de
genssler@fzi.de

Oscar Nierstrasz
Software Composition Group (SCG), University of Bern, Switzerland,
http://www.iam.uinbe.ch/~scg
oscar@iam.unibe.ch

Bastiaan Schönhage
Object Technology International (OTI), The Netherlands,
http://www.oti.com
Bastiaan_Schonhage@oti.com

ABSTRACT

Software is more and more becoming the major cost factor for embedded devices. Already today, software accounts for more than 50 percent of the development costs of such a device. However, software development practices in this area lag far behind those in the traditional software industry. Reuse is hardly ever heard of in some areas, development from scratch is common practice and component-based software is usually a foreign word. PECOS is a collaborative project between industrial and research partners that seeks to enable component-based technology for a certain class of embedded systems known as "field devices" by taking into account the specific properties of this application area. In this paper we introduce a component model for field device software. Furthermore we report on the PECOS component composition language CoCo and the mapping from CoCo to Java and C++.

Categories & Subject Descriptors

D.2.11 [Software engineering]: Software architectures — embedded systems; D.3.2 [Programming languages]: Language classifications;

General Terms

Design, Languages

Keywords

Embedded systems, Field devices, Component based software development

1. INTRODUCTION

The state-of-the-art in software engineering for embedded systems is far behind other application areas. Software for embedded systems is typically monolithic and platform dependent. Development from scratch is common practice. The resulting software is hard to maintain, upgrade and customize. Beyond that it is almost impossible to port this software to other hardware platforms. Component-based software engineering would bring a number of advantages to the embedded systems world such as faster development times, the ability to secure investments through reuse of existing components, and the ability for domain experts to interactively compose and adapt sophisticated embedded systems software. The key technical questions and challenges are:

- *Component model:* What kind of component model is needed to support modularization and re-use of software for embedded systems? Which non-functional aspects of this software (such as timing constraints) have to modeled explicitly to enable automated compositional reasoning?
- *Lightweight composition techniques:* How can component-based applications be mapped on efficient and compact code that fulfills the hard requirements imposed by the application domain?
- *Platforms and tools:* How can we increase software portability (and thus increase re-use and productivity)? What tools are needed to support efficient specification, composition, validation, and deployment of embedded systems applications built from components?

The PECOS project[1] aims to enable component-based software development for embedded systems. In order to achieve concrete results, PECOS is driven by a case study in the domain of *field devices*. Section 2 introduces the PECOS case study, summarizes the particular requirements of field devices for CBSD, and provides an example application that illustrates the PECOS working domain. Section 3 introduces the PECOS (field device) component model. In Section 4 we introduce the PECOS composition language for component based software. Examples are used, to illustrate the concepts and the mapping between the language and the model. Section 5 provides concepts for deploying the specified software to *real* devices. It describes how components are mapped to target code and how certain concepts are implemented.

2. CASE STUDY DESCRIPTION

ABB's Instruments business unit develops a large number of different field devices, such as temperature-, pressure-, and flow-sensors,

[1] Funded by the European Commission under IST Program IST-1999-20398 and by the Swiss government as BBW 00.0170. The partners are Asea Brown Bovery AG (ABB, Germany), Forschungszentrum Informatik (FZI, Germany), Object Technology International (OTI, The Netherlands), and Institut für Informatik und Angewandte Mathematik, University of Bern (UNIBE, Switzerland)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES 2002, October 8–11, 2002, Grenoble, France.
Copyright 2002 ACM 1-58113-575-0/02/0010 ...$5.00.

actuators and positioners. A field device is an embedded hard real-time system. Field devices use sensors to continuously gather data, such as temperature, pressure or rate of flow. They process this data, and react by controlling actuators like valves or motors. To minimize cost, field devices are implemented using the cheapest available hardware that is up to the task. A typical field device may contain a 16-bit microprocessor with only 256KB of ROM and 40KB of RAM.

Figure 1: Pneumatic positioner TZID

The software for a typical field device, such as the TZID pneumatic positioner shown in Fig. 1, is monolithic, and is separately developed for each kind of field device. This results in a number of problems.

- *Little code reuse*: the same functionality (e.g., Non-volatile Memory-Manager, Field-bus Driver, or control-algorithms) is re-implemented at different development locations in different ways for different field devices.
- *Plug-incompatibility*: functions and modules are implemented for a specific device without standardized interfaces.
- *Inflexibility*: monolithic software is hard to maintain, extend or customize.
- *Cyclic execution model*: Software is often implemented in several periodic tasks with different cycle times (e.g., $5ms$, $10ms$ and $50ms$). This makes it hard to incorporate sporadic and long running functions without introducing deadline misses. Furthermore it is error-prone to change software from the cyclic execution approach to process-based scheduling[7].

In order to validate CBSD for embedded systems, the PECOS project is developing the hardware and software for a demonstration field device. The task of the PECOS field device is to control a three-phase motor connected to a valve (see Fig. 2).
The motor is driven by a frequency converter that can be controlled by the field device over Modbus (an industrial communication protocol). The motor can be coupled to a valve either directly via a worm shaft or using additional gearing (4). A pulse sensor on the shaft (5) detects its speed and the direction of rotation. The PECOS board (1) is equipped with a web-based control panel (7) with some basic elements for local operation and display.The demonstrator can be integrated in a control system via the field-bus communication protocol Profibus PA (6). The device is compliant to the profibus specification for Actuators [14, 13].

2.1 Example Application

We will use the following example throughout this paper to illustrate the PECOS component model and composition language. Part of the PECOS case study is concerned with setting a valve at a specific position between open and closed. Fig. 3 illustrates three connected PECOS components that collaborate to set the valve position; the desired position is determined by other components not shown here. In order to set and keep the valve at a certain position, a control loop is used to continuously monitor and adjust the valve.

- The ModBus component works as an interface to a piece of hardware called the frequency converter, which determines the speed of the motor. The frequency to which the motor should be set is obtained from the ProcessApplication component. ModBus outputs this value over a serial line to the frequency converter using the ModBus protocol.
- The FQD (Fast Quadrature Decoder [2]) component is responsible for capturing events from the motor. This component abstracts from a micro-controller module that does FQD in hardware. It provides the ProcessApplication with both the velocity and the position of the valve.
- The component ProcessApplication obtains the desired position of the valve (Set-Point) and reads the current state of the valve from the FQD component. This information is then used to compute a frequency for the motor. Once the motor has opened the valve sufficiently, ascertained by the next reading from the FQD, the motor must be slowed or stopped. This repeated adjustment and monitoring constituted the control loop.

This example illustrates the key issues – besides the tight resource situation – concerning the field device domain: (1) **Cyclic behavior** – each component is responsible for a single piece of functionality, which is repeatedly executed (with a specified cycle time) and must not take longer than a specified worst-case execution time. (2) **Data-flow-oriented interaction** – components communicate by means of shared data. The interface of a component consists of a set of data ports. (3) **Threading** – some components are passive (i.e., cyclically invoked by a scheduler), while others (like FQD) have their own thread of control in order to react on asynchronous events or to perform long computations in the background.

3. A COMPONENT MODEL FOR EMBEDDED SOFTWARE

In order to apply component-based software development to embedded systems software, we must be precise about what we mean by a *component*. In particular, we must take care to specify how components are *structured* and *composed*, which *properties* of components are important to capture and reason about, and how a composition of components can be interpreted at *run-time*.
Here we briefly present a meta-model that reflects an architectural style [17] for embedded systems software. We also sketch how compositions of components can be interpreted by means of Petri nets.

3.1 Model Elements

Fig. 4 illustrates the key elements of the component model. *Components* have interfaces defined by a number of *ports*, and may be hierarchically composed. So-called *leaf components* are treated as black boxes, and are directly implemented in some host programming language. Composite components, on the other hand, are built by connecting the *ports* of other, existing components (leaf or composite), and expressing which ports of the constituent components are *exported* as ports of the composite (See Fig. 5.)

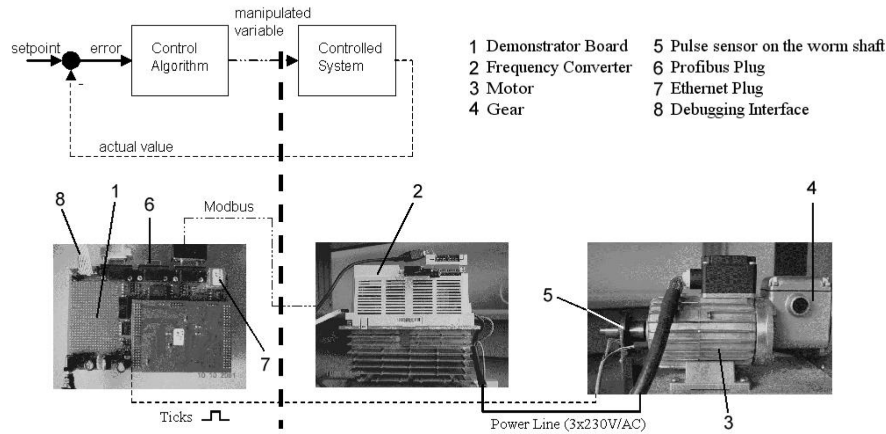

Figure 2: PECOS case study device

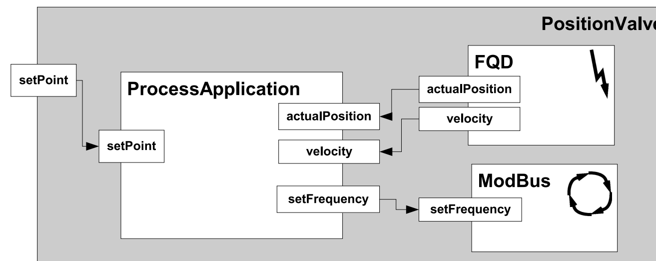

Figure 3: FQD control loop

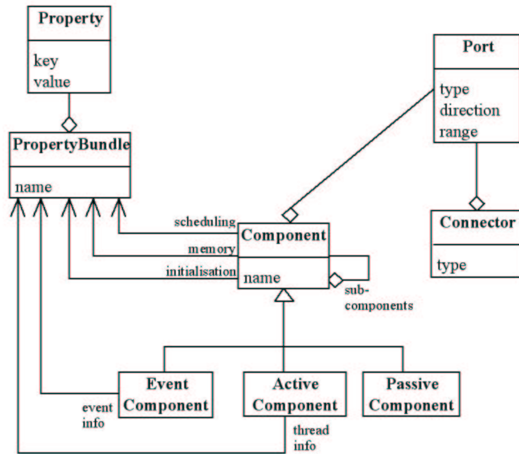

Figure 4: A Component Model for Embedded Software

Ports are shared variables that allow components to communicate with each other. Connected ports and exported ports therefore represent the *same* variable. Connectors may only connect ports of compatible *type*, *direction* and *range*.
The model expresses three kinds of components relevant for embedded systems.

Active Components (e.g., *ModBus* in figure 3) have their own thread of control. Active components are used to model ongoing or long-lived activities that cannot complete in a short cycle-time. A complete system composed of components is always modeled as a composite active component. Composite active components *schedule* their constituent components in order to meet the deadlines imposed by the real-time constraints.

Passive Components (e.g., *ProcessApplication*) have no own thread of control. They are used to encapsulate a piece of behavior that executes synchronously and completes in a short cycle time. Passive components are scheduled by the nearest active parent that contains them.

Event Components (e.g., *FQD*) are components whose functionality is triggered by an event. They are typically used to model hardware elements that periodically generate events. Typical examples are timers, used to keep track of deadlines, or devices that emit events encoding status information, such as the current rotation speed of a motor, the current temperature, and so on.

Components are characterized by their *properties*, which encode information such as timing and memory usage.

3.2 Execution model

In addition to the static structure described above, the Pecos model has an execution model that describes the behavior of a Field Device. By using Petri nets [19] to represent the execution model, we intend to reason about real-time constraints and automatically generate real-time schedules for software components.

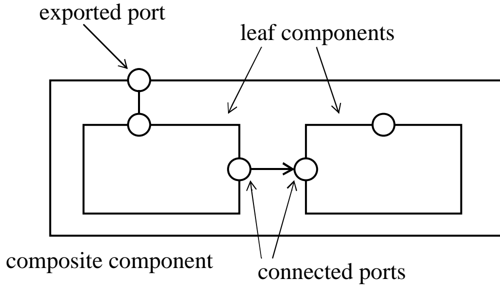

Figure 5: A Composite Component

The execution model deals with the following two issues: **Synchronization** – how to synchronize data-flow between components (esp. between components that live in different threads of control) and **Timing** – how to make sure that component's functionality is executed according to cycle times and required deadlines.
A composition of components has as many threads of control as there are active components. Each active component is responsible for scheduling the passive components under its control. We formalize the execution semantics by means of a Petri net interpretation. The details of this formalization, however, are beyond the scope of this paper. They are described in [11, 9].

4. THE COCO COMPONENT LANGUAGE

In this section we introduce the component composition language *CoCo*. The CoCo language is the syntactical representation of the model described in section 3. The language is intended to be used for 1) the specification of components, 2) the specification of entire field device applications and 3) the specification of architectures and system families. In addition, CoCo supports reuse of components and architectures and supports compositional reasoning. Last but least, CoCo serves as input for scheduler computation and code generation.
We first give an overview on the CoCo language. We describe how developers can specify components in terms of their interfaces and their behavior We subsequently show how CoCo supports the specification of system families. Finally, we sketch how CoCo specifications can be used for reasoning about functional and non-functional properties of a system.

4.1 CoCo Language Overview

Components represent units of computation and are the major means of structuring a CoCo system. CoCo supports all component types of the component model. For example, Figures 6 and 7 show a CoCo specification of our running example. There we see an active component (marked with the keyword active), an event component (keyword event) and two passive components. In analogy to the OO model, components play the role of classes. Components define a scope in the same sense as OO classes do. Components can be instantiated that is, one can create an instances of a component with a unique identity.
Components are composite when they contain instances of other components (see for example the component PositionValve in Figure 7). In PECOS, an entire application, such as a field device, is modeled as a composite component. Instances of components have a component type and a unique name within the scope of the enclosing component. All instances are created at system start-up that is, there is no "new" statement to dynamically construct new instances during run-time but all possible instances are known at compile time. This allows for a number of checks at compile-time as well as for automated scheduler generation.

```
event component FQD {
   output float actualPosition;
   output float velocity;
   property cycleTime = 100;
   property execTime = 10;
}

active component ModBus {
   input float setFrequency;
   property cycleTime = 100;
   property execTime = 10;
}

component ProcessApplication {
   input float setPoint;
   input float actualPosition;
   input float velocity;
   output float setFrequency;
   property cycleTime = 100;
   property execTime = 20;
}
```

Figure 6: The FQD control loop components specified in CoCo

```
component PositionValve{
  ModBus mb;
  FQD fqd;
  ProcessApplication pa;

   input float setPoint;

   connector c1 (setPoint, pa.setPoint);
   connector c2 (fqd.actualPosition,
                 pa.actualPosition);
   connector c3 (fqd.velocity, pa.velocity);
   connector c4 (pa.setFrequency, mb.setFrequency);
}
```

Figure 7: The FQD (Fig. 3) control loop itself specified in CoCo

Programming in CoCo is data-flow-oriented. **Ports** (e.g., "setPoint") denote data flow into or out of a component and are the only means to communicate with a particular component. One can think of the set of ports of a component as the interface to a piece of functionality that is executed cyclically or in response to a certain event in order to compute output values depending on the current input values and/or the internal state of the component. The actual behavior, however, is not specified at the level of CoCo specifications but hidden in the implementation of the component. The only information available about this implementation is the worst-case time it takes to perform the computation (property "execTime") and the interval between this computations (property "cycleTime"). These values are specified in CoCo as component properties. Ports are assigned both a data flow direction (input, output, or in-/output) and a data type. Connection of components is achieved through the use of **connectors** (e.g., connector c1 in component PositionValve). Connectors connect a list of ports defined either in the current component (like port setPoint in connector c1) or by one of the contained instances (that is, instances in the same scope). Different connectors that share a common port represent the same connection. For composite event and active components, this is only true within the scope of their ancestor component while for passive components

this also holds for ports of instances within this particular passive component.

Properties serve to specify functional and non-functional features of a component, such as init values for ports, memory consumption and worst-case execution time. They can be structured in so-called *property bundles*. These bundles group properties that semantically belong together, such as scheduling information (worst-case execution timer, cycle time). Properties can be used by tools to inspect the component in different phases of the development process (e.g., when calculating a scheduler). Properties can be set on a per-component basis and a per-instance basis.

4.2 Adding Behavior to Components

CoCo does not only support the specification of component interfaces but also provides some help for the specification of the actual behavior of components. There are two ways of adding behavior to a component. One is by composing a component out of existing components the other is by filling in code written in the target language. For the latter, CoCo provides three pre-defined hooks:

- `initialize`: Initialization code for a particular component such as init values for ports is added here. This code is executed by the run-time system upon start-up.
- `execute`: Serves to specify the actual functionality of a component that is, the algorithm that computes output values using the internal state of a component and/or input values. The time of execution of this functionality depends on the component type. For a passive component this functionality is invoked synchronously by a scheduler. For active components, this functionality is executed continuously within a separate thread. Event components (including timer components) perform their functionality when the particular event occurs that this particular event component listens to.
- `sync`: Active and event components have this additional part. The code in `sync` is executed synchronously - like execute of passive components - by a scheduler and serves to exchange data between the asynchronous thread or event handler of active components respectively event components and the synchronous outside world.

The code that can be specified here has to be valid target language code (C++ or Java). A developer can only use primitives defined by the PECOS run-time environment. This means in particular that a developer cannot start new threads or do anything else which might affect schedules. To deploy a component to a particular target platform (i.e., to generate code for this platform) all hooks of each component involved have to be filled in with appropriate target language code. The code generator adds these code fragments to the generated code.

4.3 Specifying software families with CoCo

Components serve to specify concrete pieces of a system but there is no good means yet to specify architectural styles or families of components or families of entire applications (devices). CoCo provides the concept of abstract components for this purpose. By means of abstract components one can specify a template of a system that can later be filled in with concrete components. Abstract components do not have a representation in the model as they do not contribute to the run-time behavior of field device software. They are merely a technique to simplify specification and to enable reuse of designs.

Besides the elements known from normal components, abstract components can define so-called roles. Roles are typed variation points or holes in a (micro-)architecture. Fig. 8 shows the specification of an architectural style for control loops.

```
abstract component PecosControlLoop{
  role AbstractProcessApplication PecosPA;
  role AbstractControlDevice PecosCtrl;
  role AbstractFeedBackDevice PecosFdbck;
  input float setPoint;

  connector setPoint(setPoint, PecosPA.setPoint);
  connector feedback1(PecosPA.actualPosition,
                      PecosFdbck.actualPosition);
  connector feedback2(PecosPA.velocity,
                      PecosFdbck.velocity);
  connector control(PecosPA.setFrequency,
                      PecosCtrl.setFrequency);
}
[...]
component PositionValve is PecosControlLoop{
  ProcessApplication pa is PecosPA;
  ModBus mb is PecosCtrl;
  FQD fqd is PecosFdbck;
}
```

Figure 8: Using abstract components to specify system families

We assume that a PecosControlLoop should always have an instance of sub-type of AbstractProcessApplication that plays the role PecosPA in our valve controller architecture. AbstractProcessApplication again is an abstract component that defines a certain interface (i.e., ports, properties) every process application component has to conform to. Thus, roles serve as placeholders for instances. These placeholders can also be connected by connectors as if they were normal instances. This way a developer is able to specify an entire family of applications that share a common architecture in terms of the components involved and their data-flow dependencies. To create a specific member of this family, a component has to implement the respective abstract component. Implementing an abstract component means that all roles defined by this abstract component have to be bound to suitable instances and that all connectors, instances, ports, and properties defined in this abstract component become now part of the implementing component. In our example the role PecosPA is bound to an instance of component ProcessApplication. The component ProcessApplication on the other hand is required to implement the abstract component AbstractProcessApplication.

4.4 Composition checking

In order for a composition to be valid, certain rules must be followed. Besides simple syntactic rules, that are checked by a composition language parser, some semantic rules must also be followed. These rules express requirements, that emerge from the component model. Examples include rules, like “if a component implements an abstract component, it must bind all roles” or “all mandatory ports must be bound”, etc. First order predicate logic is used to check these rules. The PECOS composition tool is able to generate a set of Prolog facts out of a composition. These facts describe the the whole system, together with all included components and their connections. Semantic rules are formulated as Horn clauses, which are checked against the generated facts.

Besides semantic rules imposed by the PECOS component model, application domain specific rules may be imposed on a specification. For example, embedded systems in a particular domain, e.g. field devices, have a specific set of requirements that could be checked using composition rules.

Finally, it may be important to impose application specific rules. Such rules could express certain requirements for debugging or re-

lease versions of the software, dependencies between components, platform specific property settings, etc. To check these rules, they are also given to the system as Prolog rules, which are checked against the generated facts.

5. CODE GENERATION: FROM COCO TO C++ AND JAVA

CoCo can be used to specify a system using the Pecos model. To be able to build a functional system out of a CoCo specification we have developed a language mapping from CoCo to target languages such as C++ and Java. This section briefly indicates the basics of this mapping.

5.1 Mapping Components

Components in the PECOS model are directly mapped to classes in the target language. Components are functional units that perform some actions by means of an execute part. Passive components within the same scope perform their execution synchronously, one at a time. Active components, however, run in parallel with the rest of the system. This is realized by mapping the PECOS model to a prioritized, pre-emptive multi-threaded system (the Pecos Execution Environment). For assigning components to a particular thread we deploy the following rules:

1. Every active and event component runs in its own thread.
2. Every passive component that is part of a composite runs in the same thread as its direct parent.

Instances of components are mapped to objects in the target language. More specifically, since instances of components can only occur inside of composite components, these instances are mapped to member variables. Instances are given the same name as the name in the specification and can be accessed through the class representing the composite component.
Ports in the PECOS model are the means to exchange data between components. In the model, three types of ports exist: input-port, output-ports and inout-ports. In the target language, ports are represented as getter and setter methods depending on the type of port. Input-ports are mapped to a get-method, output-ports to set-methods and inout-ports are mapped to both get- and set-methods. By mapping ports to getter and setter methods, we are able to hide the implementation of connectors between the ports in the methods and, at the same time, give the user an easy means of accessing the ports. As an example, Fig. 9 shows the generated code for the PositionValve component.

5.2 Mapping Connectors

Instead of only specifying stand-alone components, CoCo is designed to specify a system consisting of interacting components. Therefore a data exchange mechanism has been introduced that connects ports through connectors. A connector takes care that data at an output port is "moved" to the connected input port.
Data exchange between components can be achieved in a couple of ways. Two common approaches are: shared memory and copying values. In the language mapping we describe here, we have chosen for a hybrid approach. Within a collection of components running in a single thread data is exchanged through shared memory. This is achieved by assigning a piece of shared memory to every thread for data communication.
The Data Store that implements the shared memory is an automatically generated artifact. Inside a single thread, mapping connectors in the PECOS model to the Data Store is straightforward. The value of a connector is stored on a specified location in the Data Store. The generated get- and set-methods to exchange data on a port use indices that read from, respectively write to, that location in the Data Store. To achieve this, the generated classes use constant indices that are used to access the right value. When two ports are connected they use exactly the same index (the constant variable has the same value). This results in two ports that use the same location in the DS.
Active components together with their passive subcomponents are running in their own threads. Each thread has its own Data Store for the connectors locally to this particular thread. Connectors between ports that belong to components in different threads work differently. They exchange data by copying values from the Data Store in one thread to the Data Store in the other thread. A scheduled synchronization method (`sync`, see subsection 4.2) is used to exchange data between the different threads.
Since the behavior of the synchronize method cannot be known beforehand and it is not specified in the model either, users currently have to provide their own methods. A typical example of a synchronize method would copy data in or out of the thread's Data Store depending on the state of the component. Therefore, two utility methods are generated to aid in performing these tasks:

- `import_<portname>` : imports the value from the "outside" world into the "local" DS (input-port and inout-port only)
- `export_<portname>` : exports the value from the "local" DS to the "outside" world (output-port and inout-port only)

5.3 Executing the System

To be able to run the generated code an execution environment is necessary. In PECOS, we therefore defined the PECOS Execution Environment that abstracts from its underlying (Real-Time) OS and provides some language independent interfaces for synchronization. An execution environment for C++ and Java is defined that provides a common API for both supported target languages.
The execution environment contains a highest-priority first, preemptive scheduler. Every active component (and its passive subcomponents) in a CoCo specification are mapped to a separate thread in the target execution environment. The assignment of priorities, periods and deadlines for the tasks can be specified as a timing property of every active component. The actual schedule is computed from the values of these properties.

6. RELATED WORK

Several approaches to the composition of software from components have been proposed in the literature. An important contribution to this stems, without doubt, from the field of software architecture systems [5, 17, 8]. Architecture systems introduce the notion of components, ports, and connectors as first class representations. However, most of the approaches proposed in the literature do not take into account the specific properties of software systems for embedded devices.
In [12] van Ommering *et.al.* introduce a component model that is used for embedded software in consumer electronic devices. Koala components may have several "provides" and "requires" interfaces. Each of this interfaces defines "ports" in the sense of methods. In order to generate efficient code from Koala specifications, partial evaluation techniques are employed. However, Koala does not take into account non-functional requirements such as timing and memory consumption. Koala lacks a formal execution model and automated scheduler generation is not supported.

```
package org.pecos.generated;

import pecos.rte.component.PecosPassiveComponent;
import org.pecos.generated.DataStore;

public class PositionValve extends PecosPassiveComponent {
  public PositionValve() {
     super( "PositionValve" );
  }
  public ModBus mb = new ModBus();
  public FQD fqd = new FQD();
  public ProcessApplication pa = new ProcessApplication();
  public float get_setPoint() {
     return (DataStore.val_float[DataStore.PositionValve$setPoint]);
  }
  public void initialize() {
    mb.initialize();
    fqd.initialize();
    pa.initialize();
  }
  public void execute() {
  }
}; /* PositionValve */
```

Figure 9: Generated Java code for PositionValve component

In [18] a framework for dynamically reconfigurable real-time software is presented. It is based on the concept of so called Port Based Objects. The framework provides only a limited form of specifying a component (e.g., only rudimentary scheduling information is given, predefined port types). Furthermore the architecture is limited i.e., there is no support for composite components. The verification of a composition regarding non-functional properties such as memory consumption and schedulability is lacking too.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have presented an approach to component-based software development in the area of embedded field device software.

We have developed a component model that takes into account the specific requirements imposed by the application domain. The main features of this model are the data-flow-oriented programming style and the explicit incorporation of non-functional requirements. We have shown how components and applications can be specified in terms of the language CoCo and how these specifications are mapped onto target code. We also sketched how CoCo supports system families.

One of the goals of the PECOS project is to support all steps necessary to develop component software. This includes the component specification and composition, component implementation, model checking and deployment. During the project, prototypes of composition environments have been produced, which shall lead to a (commercial) tool. The PECOS team has decided to use the Eclipse platform (`www.eclipse.org`) as a common base for component development. The Eclipse platform (see figure 10) is an open development framework, which can be dynamically extended by plugins. As basic functionality, Eclipse support basic editing, compiling and version control of (Java) source files. In addition to this, an increasing amount of plugins is separately developed to extend Eclipse, such as C++ tooling and UML editing. In an ongoing effort, the different PECOS tools are integrated into Eclipse, to give developers a single tool, which supports the whole development cycle of component based software for field devices.

8. ADDITIONAL AUTHORS

Additional authors: Alexander Christoph, Michael Winter (Forschungszentrum Informatik, Germany), email: {`christo`|`winter`}`@fzi.de`; Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo (University of Bern, Switzerland), email: {`ducasse`|`wuyts`|`arevalo`}`@iam.unibe.ch`; Peter Müller, Chris Stich (ABB Research Center, Germany), email: {`peter.o.mueller`|`chris.stich`}`@de.abb.com`.

9. REFERENCES

[1] Embedded C++ home page. www.caravan.net/ec2plus.

[2] Fast Quadrature Decode TPU Function (FQD). Semiconductor Motorola Programming Note. TPUPN02/D.

[3] Gesellschaft für Prozessautomation & Consulting bH home page. www.gpc.de.

[4] TPTPN home page. www.diit.unict.it/users/scava/tptpn.html.

[5] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–49, July 1997.

[6] G. J. Badros and A. Borning. The Cassowary Linear Arithmetic Constraint Solving Algorithm: Interface and Implementation. Technical Report UW Technical Report 98-06-04, University of Washington, 1998.

[7] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addision Wesley, 1989.

[8] P. C. Clements. A survey of architecture description languages. In *Int. Workshop on Software Specification and Design*, 1996.

[9] S. Ducasse and R. W. (editors). Field-device component model. Technical Report Deliverable D2.2.8, Pecos, 2001. www.pecos-project.org.

[10] M. Naedele. *On the Modeling and Evaluation of Real-Time Systems*. PhD thesis, Swiss Federal Institute of Technology (ETHZ), 2000.

[11] O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, A. Black, P. Müller, C. Zeidler, T. Genssler, and R. v. d. Born. A component model for field devices. In *To Appear: Second Conference on Component Deployment*, 2002.

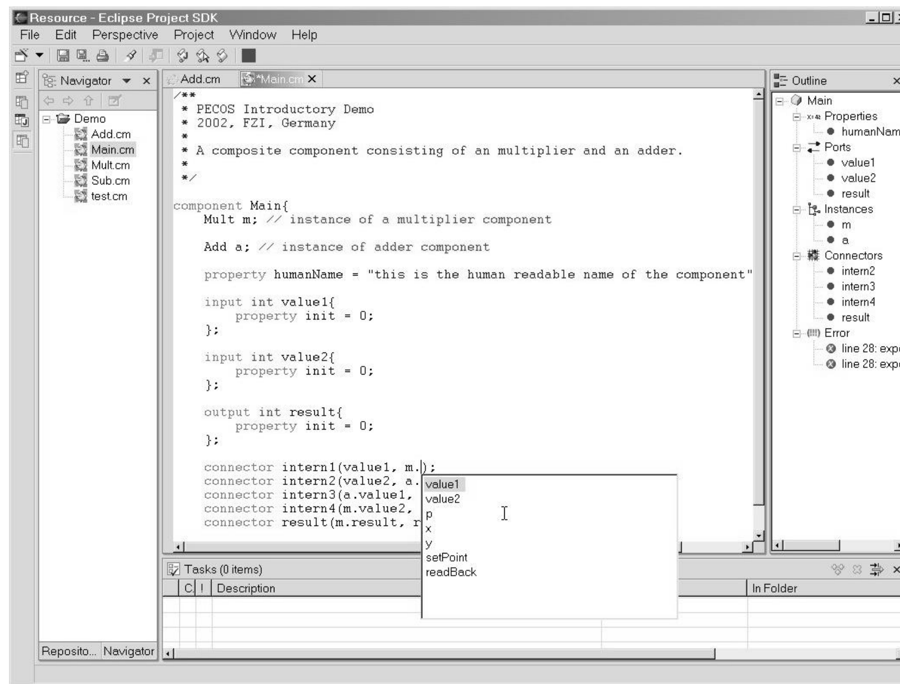

Figure 10: The upcoming PECOS toolkit

[12] R. v. Ommering, F. v. d. Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *IEEE Computer*, 2000.

[13] PROFIBUS International. *Device Datasheet for Actuators, Version 3.0*. www.profibus.org.

[14] PROFIBUS International. *PA General Requirements, Version 3.0*. www.profibus.org.

[15] B. Schönhaage. Model mapping to C++ or Java-based ultra-light environment. Technical Report Deliverable D2.2.9-1, Pecos, 2001. www.pecos-project.org.

[16] B. Schulz, T. Genssler, A. Christoph, and M. Winter. Requirements for the Composition Environment. Technical Report Deliverable D3.1, Pecos, 2001. www.pecos-project.org.

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

[18] D. B. Stewart, R. A. Volpe, and P. K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transaction on Software Engineering*, 23(12):pages 759–776, 1997.

[19] J. Wang. *Timed Petri Nets*. Kluwer Academic Publishers, 1998.