# A Component Model for Field Devices *

Oscar Nierstrasz,† Gabriela Arévalo,* Stéphane Ducasse,* Roel Wuyts,* Andrew Black‡, Peter Müller,** Christian Zeidler,‡ Thomas Genssler,†† and Reinier van den Born‡‡

Abstract. Component-based software development is becoming mainstream for conventional applications. However, components can be difficult to deploy in embedded systems because of non-functional requirements. PECOS is a collaborative project between industrial and research partners that seeks to enable component-based technology for a class of embedded systems known as "field devices". In this paper we introduce a component model for field devices that captures a range of non-functional properties and constraints. We report on the current status of PECOS, including the PECOS composition language, language mappings to Java and C++, and industrial case studies.

## 1 Introduction

The software engineering landscape is far less developed for embedded systems than for other application areas. Software for embedded systems is typically monolithic and platform-dependent. These systems are hard to maintain, upgrade and customize, and they are almost impossible to port to other platforms. Component-based software engineering would bring a number of advantages to the embedded systems world such as faster development times, the reuse of existing components, and the ability for domain experts to interactively compose and adapt sophisticated embedded systems software.

Unfortunately component-based software development (CBSD) cannot yet be applied to embedded systems development. Until now, the mainstream IT players have not paid very much attention to the (so far) relatively small embedded systems market, and consequently there exists little component-based technology or tools for embedded systems. This situation is understandable if we consider the technical obstacles that embedded systems pose for CBSD: their software must typically fulfil stringent non-functional requirements imposed by the run-time environment, such as severely limited CPU power, and memory, and by the application domain, such as hard real-time con-

* This is an extended version of a paper presented at CD 2002, Berlin.

† Software Composition Group, Institut für Informatik und Angewandte Mathematik, University of Bern, Switzerland — {oscar,ducasse,wuyts}@iam.unibe.ch — www.iam.unibe.ch/~scg/.

‡ Department of Computer Science & Engineering, Oregon Health & Science University — black@cse.ogi.edu — www.cse.ogi.edu/~black/. *Visiting SCG.*

** ABB Research Center, Germany — {peter.o.mueller, christian.zeidler}@de.abb.com — www.abb.com.

†† FZI Research Center for Information Technologies, Germany — genssler@fzi.de — www.fzi.de.

‡‡ OTI, The Netherlands — Reinier_van_den_Born@oti.com — www.oti.com.

straints. Current CBSD approaches typically assume generous run-time environments and all but ignore non-functional requirements such as timing.

The rapidly expanding market in embedded systems, however, makes CBSD for such systems not only viable but also essential. In order for industry to benefit from increasingly powerful and less expensive hardware, there is a great need to be able to develop and port embedded software more quickly and at acceptable costs. Vendors of embedded devices would benefit by being able to offer scalable product families, whose functionality could be tailored by replacing or reconfiguring software components. The key technical questions remain:

- ***Component models:*** What kind of component models are needed to support CBSD for embedded systems software?
- ***Non-functional requirements:*** How can we reason about non-functional constraints of systems based on properties of their constituent components?
- ***Tools:*** What tools are need to specify, compose, validate and compile embedded systems applications built from components?

The PECOS project* aims to enable component-based software development for embedded systems. In order to achieve concrete results within a limited frame, PECOS is driven by a case study in the domain of *field devices*, which are field deployable control devices further described in section 2. Section 3 presents a running example, and section 4 introduces the PECOS field device component model, focusing on the structural aspects. In section 5 we present the execution semantics of the component model, by translation to Petri nets. By extending this interpretation to time Petri nets [13], we intend to reason about real-time constraints and automatically generate real-time schedules. Section 6 illustrates how the component model will be applied by means of *CoCo*, a composition language [8] that can be used to configure and compose components. Finally, in section 7, we summarize the current state of the project, which is still in progress.

## 2 PECOS

ABB's Instruments business unit develops a large number of different *field devices*, such as temperature, pressure, and flow sensors, actuators, and positioners. As field devices turn into commodities, the software increasingly determines the competitiveness of the devices. As the market demands new functionality in shorter time cycles, software begins to dominate the development and maintenance costs of field devices.

Software for field devices is currently monolithic, and does not exploit CBSD for the reasons outlined above. The high cost of developing the software, the long development cycles, the high degree of architectural and implementation duplication, as well as

* Funded by the European Commission as project IST-1999-20398 and by the Swiss government as BBW 00.0170. The partners are Asea Brown Boveri AG (ABB, Germany), Forschungzentrum Informatik an der Universität Karlsruhe (FZI, Germany), Object Technology International AG (OTI Netherlands), Institut für Informatik und Angewandte Mathematik, University of Bern (UNIBE, Switzerland).

the inflexibility of current systems, offer a compelling case for attempting to apply CBSD to field devices.

## 2.1 Field devices

A *field device* is a reactive, embedded system. Field devices make use of sensors to continuously gather data, such as temperature, pressure or rate of flow. They analyse this data, and react by controlling actuators, 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 256 kB of ROM and 40 kB of RAM.

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

**Figure 1** Pneumatic positioner TZID

- ***Duplicated functionality:*** the same functionality (*e.g.*, Nonvolatile Memory-Manager, Fieldbus Driver, or FFT-algorithm) is re-implemented at different development locations in different ways for different field devices.
- ***Plug-incompatibility:*** functions and modules are implemented for a specific environment without standardized interfaces (*e.g.*, Interrupt-Driven, Port I/O)
- ***Long development times:*** the software for each project is developed from scratch without reuse of standardized architectures or components. This takes far too long.
- ***Inflexibility:*** monolithic software is hard to maintain, extend or customize.

When considering how to apply CBSD to the software for field devices, there are a number of requirements, resource constraints and current implementation practices that must be considered.

- ***Limited power:*** the available power is very limited. For some devices only 100 mW is available; this limits the choice of CPU.
- ***Limited memory:*** field devices typically have about 40 kilobytes of RAM.
- ***Standard architecture:*** the software architecture is driven by the fieldbus architecture, which is based on function blocks.
- ***Cooperating tasks:*** the software is implemented as a set of tasks, most of which run sequentially, although some are logically concurrent. Some tasks share data.
- ***Cyclic Executive:*** the software in the device runs according to a pre-defined cyclic schedule; events are also handled within the cycle.
- ***Real-time constraints:*** Parts of the software (*e.g.*, control loops, and fieldbus function blocks) must complete within real-time deadlines.
- ***Operating System:*** The operating systems used in field devices are not general-purpose, but completely dedicated. They typically offer only basic scheduling.

- ***Programming language:*** the implementation language today is C. C++ or Embedded C++ [2] may become an option if compilers become available for the low-power micro-controllers used in the devices.
- ***Static configuration:*** the device has a static software configuration, *i.e.*, the firmware can only be replaced in its entirety. Functionality is not dynamically downloadable (though this may need to change in the future).
- ***Certification:*** many field devices are used in safety-critical applications, such as chemical plants. Costly certification procedures are required for such devices.
- ***Long lifetime:*** the typical lifetime of a field device is about 10 years.

## 2.2 PECOS goals

The goal of PECOS is to enable CBSD for embedded systems by providing an environment that supports the specification, composition, configuration checking, and deployment of embedded systems built from software components.

By focusing on the field device case study, PECOS intends to deliver a demonstrator that validates CBSD for embedded systems. Specifically, PECOS intends to deliver both a *component model* suitable for characterizing software components for field devices, and a *composition environment* for expressing, validating and compiling compositions of components conforming to the model.

### Component model

The field device component model presupposes an *architectural style* for field device software. The model must therefore characterize components that conform to this style in terms of their *interfaces* and *properties*. Furthermore, the model must be capable of expressing how components are *composed* (or "connected"), and provide ways of reasoning about properties of compositions.

Specifically, the field device component model must:

- express functional *interfaces* (*e.g.*, procedural interfaces);
- express non-functional *properties* and constraints such as worst-case execution time and memory consumption;
- specify component connections and containment relations, as an architectural style;
- express *compositions* of components conforming to a style;
- support *reasoning* about the behaviour of compositions, specifically concerning *plug-compatibility*, *concurrency and synchronization*, *real-time schedules*, and *code generation.*

### Composition Environment

The composition environment enables engineers to express and validate high-level compositions of components conforming to the component model. Specifically, it must:

- support *specification* of high-level compositions (textually or visually);
- *check* that compositions conform to the constraints of an architectural style (*i.e.*, the composition rules) or of a specific application (*e.g.*, real-time constraints);

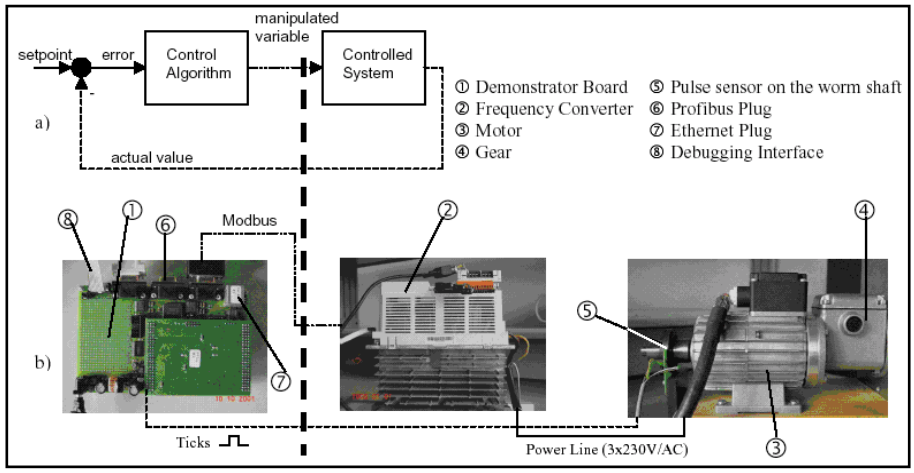


**Figure 2** Pecos demonstrator field device

- *generate* adaptation and wiring code from the high-level specification: it must "compile" the specification into executable code.

## 3 PECOS Case Study

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 figure 2). The motor is driven by a frequency converter which can be controlled over *Modbus* from the field device. 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 fieldbus communication protocol *Profibus PA* (6). The device is compliant to the profibus specification for Actuators [5][6].

### 3.1 Running Example

We will use the following example 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.* Figure 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 is responsible for interfacing 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 (hence its name). The ModBus component runs in

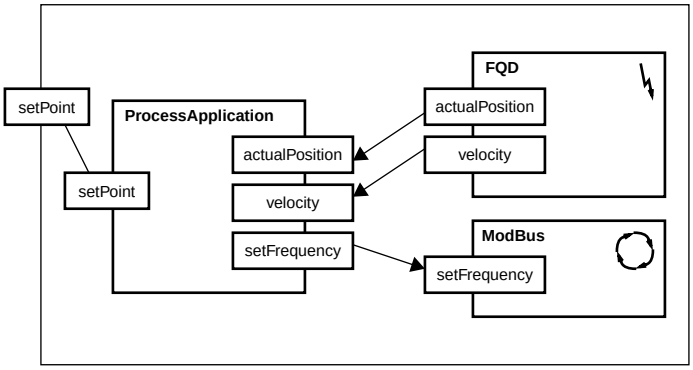


**Figure 3** FQD Control loop example

its own thread, because it blocks waiting for a (slow) response from the frequency converter.

- The FQD (Fast Quadrature Decoder [9]) 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 (setPoint) 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 several key points concerning the field device domain.

- ***Cyclic behaviour:*** each component is responsible for a single task, which is repeatedly executed.
- ***Information flow through ports:*** components communicate by means of shared data. The interface of a component consists of a set of shared data ports.
- ***Threading:*** some components are passive, while others have their own thread of control.
- ***Separate scheduler:*** control flow is separately specified by a scheduler for the composite component.

## 4 A Component Model for Field Devices

The component model presented here has been especially tailored to the domain of field devices. Although it may have broader implications for other classes of embedded systems, we do not make that claim here.

The PECOS field device component model has been defined to reflect an *architectural style* for field devices [10]. As such, we define a vocabulary of *components*, *ports* and *connectors* and the *rules* governing their composition. As in related approaches, components may only be connected if their provided and required ports are compatible [12].

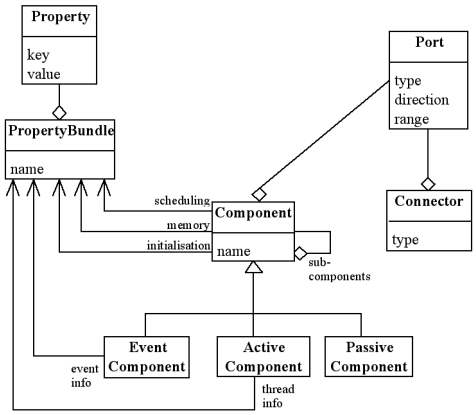


**Figure 4** The PECOS Component Model

In figure 4 we see an overview of the component model. *Components* may contain one or more subcomponents. There are three kinds of components: *passive*, *active* and *event*. Each component has three *property bundles* (for *scheduling*, *memory* and *initialisation*), and a set of *ports*. *Connectors* can be used to connect the ports of a component.

We now consider each of these elements in detail.

## 4.1 Elements of the Component Model

Components. A *component* is a computational element with a *name*, a number of *property bundles* and *ports*, and a *behaviour*. The *ports* of a component represent data that may be shared with other components. The *behaviour* of a component consists of a procedure that reads and writes data available at its ports, and may produce effects in the physical world.

A *leaf component* is a "black-box" not further defined by model, but instead directly implemented in the host programming language. It has an interface consisting of a set of ports, and properties specified by its property bundles.

A *composite component* contains a number of connected subcomponents, the ports of which form the *internal ports* of the composite. A composite component also has *external ports*, which are the only ones that are externally visible. The external ports are *connected* to appropriate internal ports. The subcomponents are not visible outside the composite that contains them.

From the point of view of a client, there is no visible difference between a composite component and a leaf component. The software of a field device can be modelled as a *component hierarchy*, *i.e.*, a tree of components, with a single active composite component at its root.

The field device domain requires three kinds of components.

- ***Passive Components*** do not have their own thread of control. A passive component is explicitly scheduled by the active component that is its nearest ancestor in the component hierarchy (its active ancestor). Passive components are typically used to encapsulate a piece of behaviour that executes synchronously and completes in a short time-cycle.

- ***Active Components*** do have their own thread of control; they are used to model ongoing or longer-lived activities that do not complete in a short time-cycle.
- ***Event Components*** are those whose behaviour is triggered by an event. They are used to model pieces of hardware that frequently emit events, such as motors that give their rotation speed, or timers that emit a timing event when a certain deadline has passed. Whenever the event fires, the behaviour is executed immediately.

Ports. A *port* is a shared variable that allows a component to communicate with other components; *connected* ports represent the *same* shared variable. A port specifies:

- a *name*, which has to be unique within the component;
- a *type*, characterizing the data that it holds;
- a *range* of values (defined by a minimum and maximum value) that can be passed on this port; and
- a *direction* ("in", "out" or "inout") indicating whether the component reads, writes, or reads and writes the data. An inout port behaves exactly like a pair unidirectional ports, one in, and the other out.

Ports of *peer components* can only be connected if they have the same type and their direction is *complementary*, *i.e.*, an in port can only be connected to an out port. *Internal* ports of a composite component can only be connected to the *external* ports if they have the same type and their direction is *compatible*, *e.g.*, an internal in port can be connected to an external in port. Internal ports may be left unconnected, so it is allowed to connect an internal inout port to an external out port.

Connectors. A *connector* specifies a data-sharing relationship between ports. It has a *name*, a *type*, and a list of *ports* it connects. (Here we consider only binary connectors.)

Properties. A *property* is a tagged value. The tag is an identifier, and the value is typed. Properties characterise components.

Property bundles. A *property bundle* is a named group of properties. Property bundles are used to characterize aspects of components, such as timing or memory usage.

### 4.2 Example revisited

Returning to the example of figure 3, we see that FQD is an event component, ProcessApplication is a passive component and ModBus is an active component. The composition will be modelled as a composite component.

FQD has "out" ports actualPosition and velocity, connected to "in" ports of the same name belonging to ProcessApplication. The in port setPoint belonging to ProcessApplication is shared with the composite component that encapsulates this composition. It is not yet connected to a compatible "out" port. Finally, the "out" port setFrequency is connected to the "in" port of the same name belonging to ModBus.

What is not yet specified is how the threads of FQD and ModBus synchronize their access to the shared ports. That is the topic of the next section.

# 5 Synchronization and Timing

Two issues must be addressed to complete the model: first, how read/write and write/write conflicts are avoided on the (shared) external ports, and second, how components are scheduled to meet deadlines.

We will do this using a Petri net interpretation of valid compositions. Using plain Petri nets we can model concurrent activities of component compositions, scheduling of components, and synchronization of shared ports. This part of the model is reasonably well-understood. Using time Petri nets we hope to reason about timing constraints, and generate real-time schedules; this topic is still under investigation.

## 5.1 Synchronization

Our Petri net procedures of the field device component model makes use of three different kinds of places and tokens. (i) *Data places* model ports; each data place has a single token representing the shared data available at that port. (ii) *Control places* are used to schedule components. Each active component has its own independent control subnet to model its schedule; there is exactly one token in each control subnet. (iii) *Event places* model the generation of an event.

A component is modelled as a Petri net fragment with a single control place that can be used to *start* it, and a single *end* place to signal that it has terminated (figure 5). When components are composed, a *schedule* must be generated that somehow moves the token from the *end* place of a component to the *start* place of the next one to be scheduled.

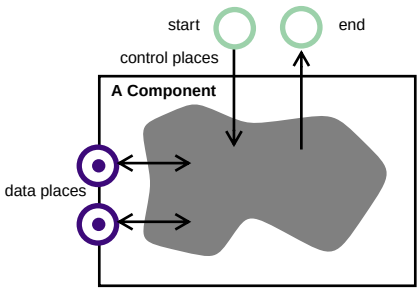


**Figure 5** Components as nets

The *behaviour* of the component is a subnet that has read and write access to its data ports. The nature of these subnets depends on the kind of component.

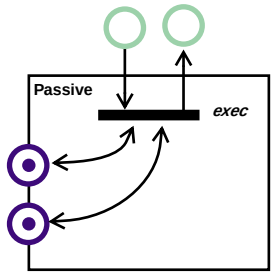


**Figure 6** Passive leaf components

Passive leaf components are particularly simple to model. Their behaviour consists of a single *exec* transition that reads or write the data places (figure 6).

Because two passive components that share a port must be serialized, synchronization problems can arise only when active components are connected to other components. Active components compete for their external data ports with their surrounding environment. To address this problem, we *split* the external ports of active components into two parts: an *outer port*, to which the outside world has free access, and an *inner port*, to which the active component has access.

These two ports are synchronized by copying the data from one to the other (depending on the direction of the port) in a special *synchronization method* (or "sync method"). This method may be generated or specially tailored. We model this by a *sync* transition that reads and writes the inner and outer ports and is triggered by the *start* control place.

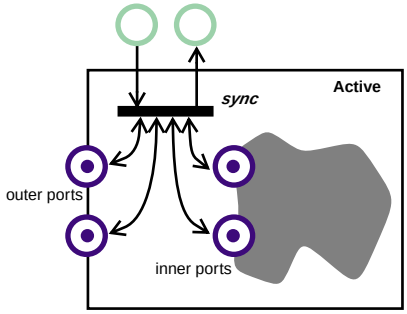


**Figure 7** Synchronization of inner and outer ports

It is important to realize that the *inner* ports are actually the *shared resources*, since they are the only ones exposed to concurrent accesses (*i.e.*, from the *sync* transition and from the internal behaviour of the active components). The *outer* ports are never exposed to concurrent accesses, because they are only accessible from the transitions of the outer control net, which contains only a single token.

The behaviour of an active leaf component is modelled as a separate control subnet consisting of a *critical section*, which may access the inner ports, and a non-critical section. The control subnet of an active component is a loop containing a single control token.

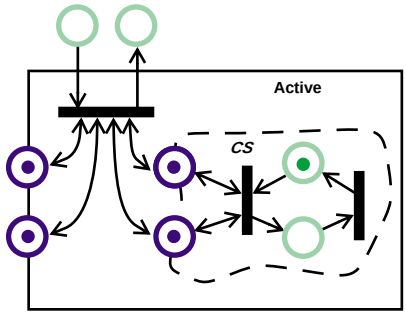


**Figure 8** Active schedule with critical section

An event component is similar to an active component, except that its control subnet does not cycle, but is triggered by an external event. To model this, we introduce an *event place* which is the target of a special transition that is fired when the event occurs.

The behaviour of an event component is implemented by its *handler*. The consequences of handling an event must, of course, be synchronized with its enclosing environment, as with active components. The event handler consumes and restores a token from a dedicated control place. This represents the fact that an event component runs in its own thread when an event is handled. At most one instance of a given handler is running at any time.

To model *composite components*, we simply coalesce all the connected data places, and we connect the start and end control places of the subcomponents according to the required schedule. In figure 10 we see that a schedule is represented by a separate control subnet. If the composition is active, this subnet will take the form of a loop with its own token; if it is passive, the subnet will have start and end control places, through which it will acquire a token when it is scheduled.

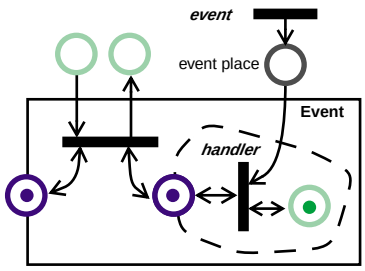


**Figure 9** Event components

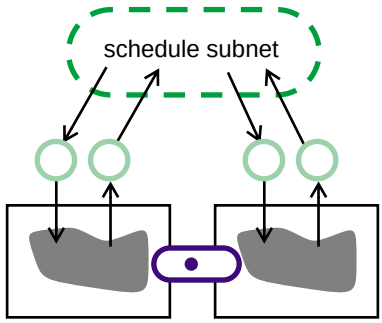


**Figure 10** Composing components

Data ports may be connected to represent data *flow*, or to represent the fact that a port of a composite component is exported from one of its constituent components. For the Petri net procedures, there is no distinction—in both cases, the connected ports represent the *same* shared variable and are therefore modelled by the same, coalesced data place.

We can see this in figure 11. Connected outer ports of all the components are each represented by a single, shared data place. This holds not only for the dataflow of velocity and actualPosition from FQD to ProcessApplication but also for the setPoint port which is visible from the outside. Inner and outer ports of active and event components, on the other hand, are *not* coalesced, since they must be explicitly synchronized.

The figure also illustrates how composite components schedule their parts. The schedule, which is triggered by the *start* place of the composite, first fires FQD, then ProcessApplication, and finally ModBus. Since FQD and ModBus have independent behaviour (*i.e.*, triggered by an event or running in a separate thread), the schedule is responsible only for synchronizing the data ports.

Note that, aside from event places, the generated Petri net is always conservative, *i.e.*, the number of tokens remains constant. This means that it is equivalent to a finite state automaton. (We could make the net strictly conservative by recycling event tokens.)

The constructed net is also clearly deadlock-free: the only conflicts between simultaneously enabled transitions occur where sync transitions compete with critical sections of active components. Since each of these transitions lock all the required data ports simultaneously, no *waits-for* cycles are possible, and hence no deadlock can arise.

## 5.2 Timing

To construct a schedule for a composition of components, certain scheduling information must be associated with each subcomponent; this includes the worst-case execution time of the subcomponent and the desired cycle time. For an active subcomponent, this information must be provided separately for the *sync* and *exec* methods. It is also be necessary to specify a (partial) ordering for the execution of the subcomponents in a composition.

The simplest form of schedule is a cyclic executive, in which components are wired together directly, such that each passes control to the next. Such schedules are very efficient, but are feasible only for the very simplest of compositions. Such a schedule can be represented in the Petri net model by inserting transitions between the end place of one component and the start place of the next.

If a more complex schedule is desired, we introduce a separate scheduler network, which makes scheduling decisions and then implements them by placing tokens on the start places of the components. This arrangement also has the advantage that the sched-

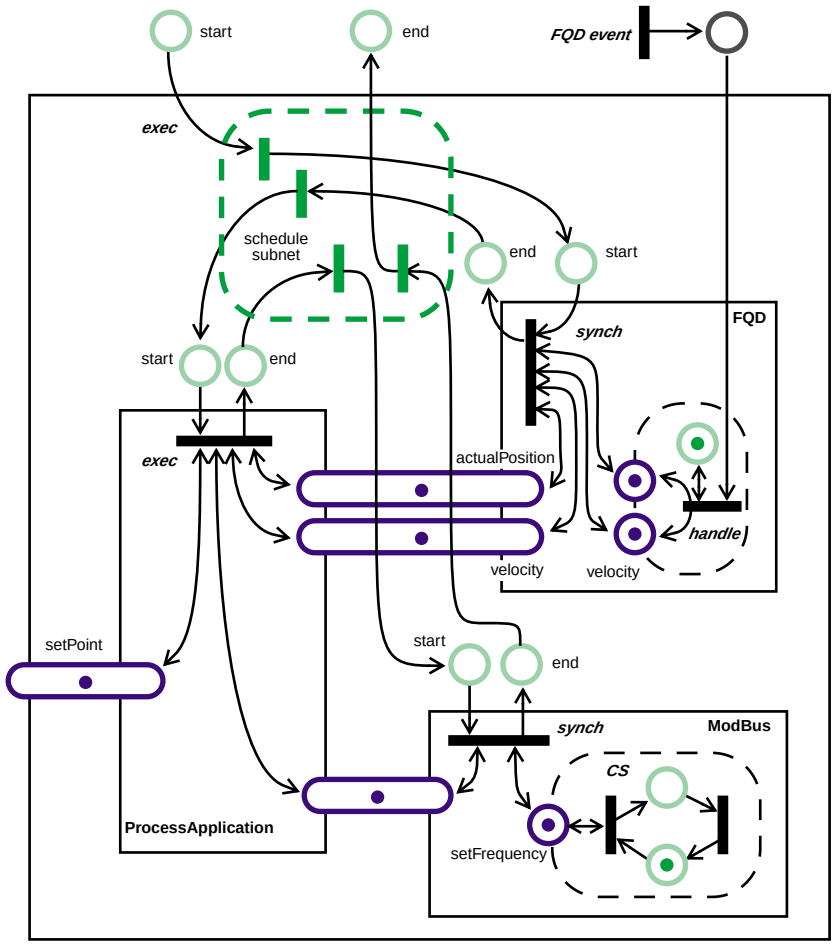


**Figure 11** Petri net model of control loop example

uler is factored out of the design; it is a subnet separate from that of the components themselves, and with a standard interface, as shown in figure 10. These are exactly the conditions that promote reuse.

So long as the schedule can be computed entirely statically, there is no need to explicitly represent the scheduling information for the components: it is represented implicitly in the schedule itself, but that schedule can be calculated off-line. However, it may still be desirable to verify the scheduler produced in this way. We have been investigating the use of time Petri nets [14] to represent the schedulers, and various Petri net tools such as Poses++ [3] and TPTPN [11] to assist in their analysis. We have also been investigating the use of hierarchical constraint solvers (such as Cassowary [1]), to capture the timing requirements and partial ordering of components and check the feasibility of schedules, or even to generate them when this is possible.

If static scheduling is infeasible, as it often will be in applications in which the average case is very much better than the worst case, then dynamic scheduling will be necessary. This means that the scheduler must make scheduling decisions while the application is running, in real time. As a consequence, the scheduling parameters for all of the components that must also be available in real time.

In his PhD thesis [4], Naedele proposes a modular approach to capturing this information. Associated with the Petri net that represents the structure of each component is a timing subnet that represents the scheduling information. The timing subnet is also responsible for mediating communication between the component and the scheduler; this allows the scheduler to communicate with the components using a broadcast network, rather than requiring separate communication channels with each component. This greatly simplifies the wiring between the scheduler and the components, in the same way that a broadcast network like Ethernet simplifies inter-computer communication, compared to point-to-point wiring. In particular, the structure of the scheduler does not have to be modified when an additional component is added to the application; such a change is instead captured by adding additional tokens containing scheduling information. This arrangement promotes modularity, and permits different kinds of schedulers to be plugged in.

## 6 The Composition Language CoCo

CoCo [8] is the syntactic representation of the model described in section 4. The language is intended to be used for both the specification of *components* and the specification of field device *applications* built as compositions. These specifications are then used as a basis for reasoning about applications and their constraints, computing schedulers that meet these constraints, and generating code.

Although a complete description of CoCo is beyond the scope of this paper, a brief summary will help us to explain how the field device component model is applied to develop applications.

CoCo supports the key elements of the component model introduced earlier: *components*, *properties* and *ports*. In addition, *instances* and *connectors* are used to build composite components.

```
event component FQD
{
  out float actualPosition;
  out float velocity;
  property cycleTime = 100;
  property execTime = 10;
}
active component ModBus
{
  in float setFrequency;
  property cycleTime =100;
  property execTime =50;
}
component ProcessApplication
{
  in float setPoint;
  in float actualPosition;
  in float velocity;
  property cycleTime =100;
  property execTime =20;
  out float setFrequency
}
```

**Figure 12** FQD control loop in CoCo

In figure 12 we see our running example as it would be implemented in CoCo. CoCo distinguishes the same component types as component model does. In our example we see an active component (marked with the keyword active) an event component (keyword event) and two passive components.

```
component PositionValve
{
  in float setPoint;
  ModBus modbus;
  FQD fqd;
  ProcessApplication processapplication;
  connector c1 (setPoint,
       processapplication.setPoint);
  connector c2 (fqd.actualPosition,
       processapplication.actualPosition);
  connector c3 (fqd.velocity,
       processapplication.velocity);
  connector c4 (processapplication.setFrequency,
       modbus.setFrequency);
}
```

**Figure 13** A composite component in CoCo

Components model units of computation. In analogy to the object model, components play the role of classes. A component defines a scope in the same way as a class. Components can be instantiated, that is, one can create an instance of a component with a unique identity.

Composite components like PositionValve contain instances of other components. An entire application is modelled as a composite component. Instances of components have a component type and a name that is unique within the scope of the enclosing component. All instances are created at system start-up, that is, there is no new statement to dynamically create new instances. Hence all instances are known at compile time; this allows for a number of static checks as well as for automatic scheduler generation.

Ports, such as setPoint) denote data flow into or out of a component and are the only means to interact with a particular component. On can think of the set of ports of a component as the interface to a procedure 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 behaviour of this “procedure” is not specified by CoCo but is hidden in the implementation of the component; the only information available about it is the worst-case time taken to perform the computation (property execTime) and the interval between these computations (property cycleTime). Ports are assigned both a data flow direction (in, out, or inout) 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).

A system specified in CoCo can relatively easily be translated into target languages such as C++ or Java. The component structure from the CoCo specification can be mapped directly to an identical class structure in the target language. Any local functionality of components, as specified by the user in the target language, can be simply incorporated. Instances map to statically initialized instance variables, and connectors represent shared instance variables in the enclosing object. Ports map to set- and get-methods that read from or write to these shared instance variables, as determined by the

connectors. Every read/write operation to the same data location is serialized so there is no need for locking.

When generating code from a CoCo specification, special attention needs to be given to the efficiency, measured both in execution time and in memory consumption, because of the requirements imposed by the field device domain. More information on these issues can be found in the relevant Pecos deliverable[7].

## 7 Status and Future Work

ABB's Business Unit "Instruments" (BUI) develops a large number of different field devices, such as transmitters for measuring temperature, pressure and flow, and actuators and positioners for transmitting a torque to a valve. There is a general trend to move more and more intelligence from the higher levels of the control system to the level of the individual field devices. Such intelligent field devices provide built-in features for measuring, tracking and reacting to instrument performance, status and maintenance history. For example, the new Asset Optimization approach supports preventive maintenance through early detection of degrading functionality. Furthermore, features to support configuration, maintenance and rapid set-up, using remote access as well as local user interfaces, are becoming increasingly important for the market success of field devices.

By using the PECOS CBSD approach for developing field-devices ABB's BUI expects to:

- serve the field device market with value-added features in a cost-effective way (*i.e.*, by adding new features like advanced maintenance triggers);
- break up the close link between hardware and software and therefore encourage the reuse of components, not only within but across families of field-devices; and
- drastically reduce the testing phase when modifying devices (*i.e.*, when adding a new component or exchanging an existing one).

In this paper we have presented some intermediate results of the PECOS project. The component model developed for PECOS addresses the requirements identified for the case study outlined above. Ongoing activities include: (i) formalization of the component model, (ii) investigation of various techniques, such as time Petri nets and hierarchical constraint solvers, to generate real-time schedules, (iii) implementation of components to support the PECOS case study demonstrator, (iv) implementation of the language mapping to generate executable code from CoCo specifications of component compositions.

## 8 References

[1] Greg J. Badros and Alan Borning, "The Cassowary Linear Arithmetic Constraint Solving Algorithm: Interface and Implementation," Technical Report, no. UW Technical Report 98-06-04, University of Washington, 1998.

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

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

[4] Martin Naedele, “On the Modeling and Evaluation of Real-Time Systems,” Ph.D. thesis, Swiss Federal Institute of Technology (ETHZ), 2000.

[5] PROFIBUS International, PA General Requirements, Version 3.0 (www.profibus.org)

[6] PROFIBUS International, Device Data Sheet for Actuators, Version 3.0

[7] Bastiaan Schönhage, “Model mapping to C++ or Java-based ultra-light environment”, Pecos Deliverable D2.2.9-1, www.pecos-project.org

[8] Benedikt Schulz, Thomas Genssler, Alexander Christoph, Michael Winter, “Requirements for the Composition Environment”, Pecos Deliverable D3.1, www.pecos-project.org

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

[10] Mary Shaw and David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

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

[12] Rob van Ommering, Jeff Kramer, Jeff Magee, “The Koala Component Model for Consumer Electronics Software”, IEEE Computer, March 2000, Vol. 33, No. 3, pp. 78-85.

[13] Jiacun Wang, *Timed Petri Nets*, Kluwer Academic Publishers, 1998.

[14] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. IEEE Transactions on Software Engineering, 17(3), pp. 259–273, 1991.