Software Component Model for Field Devices

Gabriela Arévalo, Stéphane Ducasse, Peng Liang
Oscar Nierstrasz, Roel Wuyts

July, 2002
PECOS-D2.2.8

Abstract
The domain of embedded devices and specifically the domain of field devices urgently needs new solutions to support the development of embedded software. While component-oriented programming seems a promising approach, general-purpose component models do not address the specific constraints of field devices, namely time response and memory consumption substitutability of the components. This report describes a specific component model that addresses the constraints of field devices. This version of the field device component model is structurally stable, and is consistent with the language mapping and the PECOS demo. The model enables composition rules to be introduced. Synchronization and scheduling issues are partially specified.

This research has been partially funded by European CORDIS Community Research project (EU), PECOS: Pervasive Component Systems (IST-1999-20398).
1 Introduction

The domain of embedded devices and specifically the domain of field devices urgently need new solutions to support the development of embedded software. Although the capabilities of embedded devices are increasing rapidly, their responsibilities increase likewise. Distributed embedded devices (intelligent field devices, smart sensors) not only acquire but also pre-process data and run more and more sophisticated application programs (control functions, self-diagnostics, etc.). The challenge of this shift is that the software needs to follow. Here the story is less positive: the software engineering techniques that are typically employed are lagging far behind software engineering techniques for mainstream applications. Currently software in embedded devices is written in assembly language or C, in a monolithic fashion, with a typical development time of two to three years.

The reasons for this are two-fold. The first reason is due to the specific context of embedded devices, which imposes constraints of power consumption and simple hardware. The second reason is that, up until a couple of years ago, the market for embedded devices was relatively small, and therefore the field was neglected by providers of development tools and methods who focused instead on desktop applications. For example, operating systems or development environments are much harder to find for embedded systems. The goal of the PECOS (PErvasive COmponent Systems) project is to apply solutions for component-based software engineering (CBSE) in the context of embedded systems. As with desktop applications, the overall goal is to reuse more in the way of software components, designs and architectures, and thereby improve quality and reduce development time.

In section 2 of this report we present and motivate the specific requirements the field device component model is intended to address. In section 3 we informally present the structural and execution aspects of the component model, and we illustrate the model by means of examples. Section 4 describes the formalization of component model using Petri Nets. Section 5 presents prior and related work. The report concludes in section 6.

2 Specific Requirements

Field devices are embedded reactive systems. A field device can analyze temperature, pressure, and flow, and control some actuators, positioners of valves or other motors. Field devices impose certain specific physical constraints. For example, a TZID (a pneumatic positioner) works under the following severe constraint: the available power is only 100 mW for the whole device. This limits severely the available CPU and memory resources. The TZID uses a 16 bit micro-controller with 256k ROM and 20k RAM (on-chip), and communicates using the Fieldbus communication stacks (an interoperability standard for communication between field devices). The device has a static software configuration, i.e., the firmware is updated/replaced completely, and there is no dynamically loadable functionality. Because of the specific nature of field devices, field devices are subject to three kinds of constraints: physical, software development and deployment.

Physical Constraints.
As a result from the physical constraints (especially the very stringent power consumption requirements), the runtime environment and the software in the context of the PECOS project are subject to the following constraints.

1. One processor: all the components composing a field device are running on one dedicated processor, optimised for overall power consumption. The speed ranges from a few MHz to up to several 10th of MHz on more powerful field devices. Extra functionality, like a floating point unit, is often stripped. On the other hand, the processors typically have some RAM, non volatile memory and a great deal of I/O functionality such as timers, and general purpose I/O on-chip.

2. Limited amount of slow memory: field devices typically have about 20 kilobytes of memory.

Software Development Constraints.
In addition to the physical constraints, field device development imposes the following constraints.

1. Partial State Automata Description: Some, but not all parts of a field device can be described by state-automata.

2. Cyclic execution model: The software in the device runs according to one fixed, predefined schedule that runs continuously. Exceptions are also handled within those cycles.
Deployment Constraints.

1. **Operating System**: The operating systems used in field devices are not mainstream, but completely dedicated. Because they are optimised for having a minimal CPU load, they typically offer only basic scheduling. However, RT-OSs have support for concurrency and synchronization.

2. **One single program image**: after assembling the different components that compose a field device, the software for the field device forms one single piece that is burnt into non-volatile memory.

3. **No dynamic change**: At run-time (after the field device is initialized) there is neither dynamic memory allocation, nor dynamic reconfiguration.

4. **Single language per application**: a component is created in a single language like C or C++. The overhead of converting data types from one language to another’s representation at runtime is simply too big.

Requirements.

To cope with these constraints a component model should be defined that is used for:

- Structural checking (checking of well-formedness of components and structural checks on the composition of components), and
- The checking of non-functional requirements.

After analyzing the domain with the domain experts over several iterations, the following non-functional requirements were identified as being most critical:

- **Timing/scheduling**: executing the functionality of a component takes a certain time, and during one cycle of the scheduler in the device components may need to be run several times. Hence, when substituting one component for another, it needs to be checked whether the new component fits these requirements. This is closely related to the scheduling of components, and the problem of (semi-) automatically deriving a schedule for a field device in which the components have been documented with information regarding their runtime timing behaviour.

- **Memory consumption**: One has to be sure that a certain configuration of components fits the memory requirements of the field device that is being built.

Note that we do not explicitly address the non-functional requirement of power consumption. The reason is that power consumption is directly related to timing, and support for timing immediately implies support for power consumption. Once timing is supported, the power consumption can be calculated using the frequency of the CPU. Then what-if scenarios based on changing the CPU frequency can be supported as well.

Another important aspect that has to be supported by the model is the handling of synchronization and concurrency. Since solutions favored by desktop-oriented component models are too costly to be employed in a field device, the model has to propose an architecture that makes sure that synchronization errors cannot occur. Lastly, it needs to be possible to generate code skeletons from the model where the component developer can implement the functionality of the component. However, the developers need not to be concerned with synchronization or locking issues; the generated code needs to provide enough functionality in the form of an API so that the necessary data modeled by the component is usable in the implementation.

The current constraints, in particular the fact that only few parts of a field device are described by state-automata coupled to the fact that the component time and memory consumption characteristic depend on the hardware, OS, current consumption and memory availability exclude formal proofing techniques. Instead, the model is based on experimentally obtained figures about the runtime behavior of components. How to obtain these figures is thus a crucial part for the usability of the model to check non-functional requirements. The process to get these figures and how to use them are the topics of new reports.

So, when we recapitulate these points, a component model for field devices should provide solutions for the following problems:

1. handling synchronization in the context of field devices,
2. checking timing information and checking or generating scheduling information, and checking memory consumption to make sure the software can run on a certain memory configuration.

### 3 Field Device Component Model-V
This section starts with an overview of the model, and then presents the model from a structural and an execution point of view.

### 3.1 Model Overview

The Field Device Component Model-V consists of components, ports and connectors. It enforces an execution model where components can run concurrently, but where they can only communicate data on their ports during synchronization events determined by a scheduler. Before we introduce the model in full detail, we give a structural overview. The structure is depicted in Figure 1. The core entity is the component. Every component can contain one or more connected sub-components, has property bundles (scheduling and memory), and a set of ports. Ports denote data that is available to share with other components. Connectors are used to model actual data sharing between specific ports on specific components. There are different kinds of components: passive, active, and event.

![Figure 1: PECOS Component Model diagram](image)

### 3.2 Definitions of the model entities

**Component.** A component is the main entity in this model. Components are used to organize the computation and data into parts that have well-defined semantics and behavior.

Every component has a name, a number of property bundles and a number of ports. The behavior of a component is not modeled explicitly. Instead, the component is viewed as a black box where some internal process executes that reads and writes data from the ports. A component can have connected subcomponents, in which case it is called a composite component. The subcomponents of a composite component are not visible outside the composite component. This also means that there is no visible difference between leaf components and composite components. As described in Section 3.6 a field device is modeled as a component hierarchy, i.e., a tree of component, with an active composite component at its root. When a component has no subcomponents, it is sometimes called a leaf component.

The ports of a component can be connected to ports of other peer components. Ports offer the sole mechanism for a component to interact with other components (the outside world).

Note that we generally refer to “components” even though we often intend “instances of components”. Composite components are composed of instances of other components.
Based on the experience of the domain experts, we differentiate between the following kinds of components.

- **Passive Component.** A passive component does not have its own thread of control. It is explicitly scheduled by the “active ancestor” that contains it (see Section 3.3). Passive components are typically used to encapsulate a piece of behaviour that executes synchronously and completes in a short time-cycle.

- **Active Component.** An active component is a component with its own thread of control. Active components are typically used to model either very fast or very slow activities (such as reading out hardware registers or writing to slow memory).

- **Event Component.** An event component is a component whose behavior is triggered by an event. Certain pieces of hardware frequently emit events, such as motors that give their rotation speed. To model this, the model includes event components. Whenever the event fires, the behavior is executed immediately.

**Port.** A port is a shared variable that enables a component to be connected to another component (through a connector). A port specifies the following information:

- the name of the port, which has to be unique within the component;
- the type of the data passed over the port;
- the range of values (i.e., between a minimum and maximum value) that can be passed on this port; and
- the direction of the port: ports can be unidirectional (‘in’ or ‘out’) or bi-directional (‘inout’).

A port can only be connected to another port having the same type and complementary direction.

**Connector:** A connector describes a data-sharing relationship between ports. It has a name, a type (that has to be compatible with the port types), and a list of ports it connects.

Note that connections can only exist between ports that are on a component (on its inside) and/or any of its direct sub-components (on their outsides).

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

**Property Bundle.** A property bundle is a named group of properties. Typically, sets of properties are used, for example to give all the information for some aspects of a component, such as timing or memory consumption.

### 3.3 Parent and Scope chain

The component structure implied by the model is always hierarchical. The top is typically formed by an active composite component that contains a number of subcomponents\(^1\). Every one of these subcomponents can again contain passive or active subcomponents. Because of this hierarchical structure we introduce a scope for components.

First of all we define the parent for a component X. The parent of X is the immediate composite component within which X is nested.

Then we define the scope chain for a component X to be the list of all the (direct and indirect) parents of X, ordered according to the shortest distance from X. So, when we have a composite component v, that contains a subcomponent w, that in turn contains a subcomponent x, then the scope chain for x would be \{w, v\}, in this order.

We define the active ancestor of a component X as the first active component in the scope chain of X.

### 3.4 Execution Model

The Section 3.3 defines the structural entities of the model. In this section we discuss the execution model. The execution model describes that components can run concurrently, but can only communicate data over their ports one component at a time.

Data ports are shared variables that must be synchronized in the presence of concurrency. Each active component determines when its subcomponents may access their data ports by “scheduling” one subcomponent at a time. Since each active component has a single thread, this guarantees that a

---

\(^1\) If not, then the top component has no sub-components. For the model this degenerate situation poses no problems. In the explanation we do not consider this situation to be very relevant.
subcomponent will have exclusive access to its data ports when it is scheduled. Similarly, an active component may access its own external data ports when its active ancestor schedules it. At this point, the thread of the active component and that of its active ancestor must be synchronized to maintain consistency.

To explain the execution model and to show how the components synchronize the data on their ports we use the following metaphor based on tokens.

- A component can only access the data on its ports when it has a token. The rest of the time it cannot access its ports, even if it is an active or an event component.
- There is one single token for each active component to schedule the components it contains.

Note that a token is only used to determine which component has access to the data on its ports. It is not linked to the execution of components. Components that execute concurrently synchronize their access to the data based on scheduling sequences.

The way a token is passed around, and guaranteeing that there is only one token at any moment per active component, is the work of the scheduler. First we explain what each component can do while it has the token, and then we discuss how the scheduler distributes the token.

### 3.4.1 Component Communication Synchronization

The communication of parallel components requires a synchronization mechanism. In this section we describe how the presented model is executed. The different components behave differently while they get a token. A component at the top of the field device is always an active composite component in which are nested other components. So a component hierarchy can be seen as a tree where each active component heads a sub-tree which can be empty. All components within a sub-tree run in the same thread and are controlled by the active component at its head. At the points where sub-trees are connected, synchronization is exercised by limiting the sub-tree access to its environment.

**Passive component.** While a passive component gets the token, its internal behavior is executed. The component can read and write the data specified by its ports. Because this component has the token, it is the only one that has access to the data and so it does not need to be synchronized.

**Active component.** An active component runs concurrently. While an active component is running, it may control and schedule its own sub-components, but it does not have a priori access to its own external data ports. When the active component gets the token from its active ancestor, and only then, does it have the opportunity to synchronize its internal ports with its external ports. It can decide which data have to be synchronized and propagated to its internal components. The rest of the time the active component can be left running safely and independently of its peers. Note that, from an implementation point of view, the point in time when the active component gets its token, its internal thread must be synchronized with the thread of its active ancestor. The details of this synchronization are left to the code generation, or, in special circumstances may be tailored by the developer.

A token is passed around between the activities that are controlled by an active component. These activities are for an active component A:

- A’s own behavior,
- the execution of all passive components that have A as active ancestor i.e., any passive component that in the partial sub-tree rooted by A, and
- the synchronization of all active components that have A as active ancestor, i.e., a single token is passed between the active sub-components, i.e., any active component that is the root of a sub-tree immediately under above mentioned sub-tree.

**Event component.** An event component encapsulates an action that has to be executed every time an event occurs. While the event component has a token, and only then, it may access the data on its ports. Note that raising an event does not imply that the component gets the token.

### 3.5 Example

In this section we give an example to illustrate the behavior of a model. Note that we assume that a certain schedule exists. How to manually express schedules, or derive them from a model is described in PECOS-D2.2.6-2.

The set-up of this example is given in Figure 2. In this example, component A has three sub-components (r, s and t). Component r is a passive composite component that contains a passive leaf component m and an event leaf component n. Component s is an active composite component, that contains a passive sub-component v and an active leaf component w. Component t is just a passive leaf component. We have given these components ports, numbered from 1 to 14, and shown the connectors for these ports. Note that we have omitted a lot of information from the picture that is available in a
complete model description (such as directions on ports, property bundles of components, etc.). The reason is that we are interested in showing how the component gets scheduled, i.e., how the token is passed by the components.

![Diagram of component hierarchy](image)

**Figure 2: Example of a composite active component containing leaf and composite sub-components. Note that this is not a complete model (it omits some information for clarity)**

Note that the example contains two composite components that are active (A itself and s). So, each of these components defines its own scheduler to schedule its sub-components. Like we said before, every component is scheduled by the first composite active component in its scope chain, so in this case this means that the scheduler of s schedules components v and w. All other components (including s) are scheduled by the scheduler of A.

Suppose in this example that the schedule for the component A is to schedule its sub-components in the following order: r, m, n, s, and t. Suppose also that the schedule for s schedules its sub-components in the order: v followed by w.

This example then results in the parallel execution of the following tasks:
1. r.exec, m.exec, n.sync, s.sync, t.exec
2. s.exec, v.exec, w.sync
3. w.exec
4. n.exec

Note how each task runs basically in its own memory space, which is synchronized with the rest of the device when a sync method is executed. For task 1 this would be when A.sync is called (which is not shown, since A.sync runs in the task of A’s parent). Task 2 is the task corresponding with component s. Hence the ports 8, 9, 10 and 13 are synchronized when s.sync is called in task 1. Outside this synchronization, the internal parts of the ports are used while the activities in task 2 execute. Since these activities happen sequentially, no locking of the internal values of the ports is necessary. The same is true for the other tasks: activities run sequentially within a task and use the internal values of ports.

Note that the fourth task is special in the sense that it only executes whenever the event that is watched by the component is triggered. When the trigger occurs, n.exec is executed and (like with other components) only has access to the internal data ports.

### 3.6 Field Device

In the previous sections we described the component model for field devices. Now we are ready to use this model to describe what a field device looks like.

Based on the feedback from the domain experts, we model a field device as an active composite component that contains a number of predefined components that are common to all field devices.

---

2 Other schedules are of course possible. Hence at least some information will need to be given by the component developers, such as the ordering of the components and the timing requirements. From this information the schedule will then be checked or generated, see PECOS-D2.2.6-2.
considered in the project at this moment. When the scope broadens, other field devices might be considered. A field device has the following set of constraints:

- A field device is an active composite component.
- It contains the following 5 sub-components:
  - Human Interface component. This component models the display on a field device.
  - Non-volatile memory component. Field devices have a non-volatile memory for storage and retrieval of settings.
  - Bus component. This component abstracts the field device from the actual bus used. Hence different field device buses such as Profibus or FieldBus Foundation should be exchanged without impacting the complete component.
  - Device component. This component describes the actual hardware device(s) used and wraps it as a component.
  - Input-Output-Controller Component. This component fulfils two roles: controlling the device and converting raw data from the hardware to less hardware-specific forms. For example, it can scale raw data from a temperature controller to degrees Celsius.

This list is still very crude. As more devices are modeled, ports and connections that are common to all devices will be added.

3.7 Discussion

**Deadlock avoidance.** As described in [CES71], there are four necessary and sufficient conditions for the occurrence of deadlock:

1. **Serially reusable resources:** the processes involved share resources that they use under mutual exclusion.
2. **Incremental acquisition:** processes hold on to resources already allocated to them while waiting to acquire additional resources.
3. **No pre-emption:** once acquired by a process, resources cannot be pre-empted (forcibly withdrawn) but must be released voluntarily.
4. **Wait-for cycle:** a circular chain of processes exists such that each process holds a resource that its successor in the cycle is waiting to acquire.

The nature of the field device model means that the second condition can never occur. Since the components are scheduled linearly, they cannot hold on to resources. When their timeslot passes, they are not active anymore. Moreover active and event components work on local copies of the data. The same motivation holds for the fourth condition: there is only one component active that can use the data at any time. Hence there is never a chain of processes that is waiting. Hence we can conclude that the field device component model is deadlock-free.

**Synchronization.** The presented execution model ensures that sibling active components cannot communicate directly. They always talk through the data buffers of their common active ancestor and are automatically synchronized by the token mechanism. Of course, this does not come for free: they need to wait for the token to pass by.

4 Formalizing Component Model-V with Petri Nets

In this section we describe a formalization of the model using Petri Nets. This gives a concise description of the semantics of the model besides the more informal explanation given earlier.

**4.1 Petri Nets: Basic Definition**

A Petri Net is an abstract, formal model of information flow [Pete77]. Petri Nets were developed to describe and analyze the flow of information and control in systems that can manage asynchronous and concurrent activities. The most thoroughly explored systems are those where events can occur concurrently and where there are constraints on the concurrency, precedence, or frequency of these events.

Formally, a Petri Net is a five-tuple \((P,T,I,O,M)\) where:

- \(P\) is a set of places
- \(T\) is a set of transitions
- \(I\) is an input function: \(I : T \rightarrow 2^P\) (The function identifies the places leading into a transition)
• $O$ is an output function: $O: T \rightarrow 2^P$ (The function identifies the places leading out of a transition)

• A marking $M: P \rightarrow \mathbb{N}$ (A mapping of places to non-negative integers).

A transition is enabled if each of its input places has at least one token. A transition may fire if it is enabled and the effect is to remove a token from each input place and add a token to each output place. A Petri Net is usually associated with an initial marking, or sometimes a set of initial markings. This way the starting state(s) are identified. Given a net and an initial marking, it makes sense to ask what are the possible states (markings) that can be reached.

The state space of a Petri Net is the set of all markings. A marking $M'$ is immediately reachable from marking $M$ if the firing of some enabled transition in $M$ yields $M'$. The reachability set is the set of all markings that can be reached from some initial marking $M$.

### 4.2 Applying Petri Nets to Component Model-V

Two issues in the model must be addressed: how read/write and write/write conflicts are avoided on the (shared) external ports, and 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. We will use an extended version of Petri nets to reason about timing constraints.

#### 4.2.1 Mapping Components Description into a Petri Net

We need to translate the different components and their compositions in terms of a Petri Net representation. Thus, we can check the concurrency, the scheduling and the synchronization between components using the properties provided by the Petri nets formalism.

The Petri nets representing the field device component model make use of three different kinds of places and tokens:

- **Data places** models ports. Each data place has a single token representing the shared data available at that port.
- **Control places** schedule components. Each active component has its own independent control subnet to model its schedule; there is exactly one token in each control subnet.
- **Event places** model the generation of an event.

A component is modeled as a Petri Net fragment with a single control place which can be used to start it, and a single end place to signal that it has terminated (Figure 3). When components are composed, a schedule must be generated that introduces transitions to move the token from the end place of a component to the start place of the next one in the schedule.

![Figure 3: Components as Petri Nets](image)

The behavior 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.

**Passive leaf components** are particularly simple to model. Their behavior consists of a single exec transition that reads or writes the data places (Figure 4). 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 5).

The behavior of an active leaf component is modeled 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. In Figure 6, we show only one critical section and one non-critical section but the control subnet may be refined to consist of multiple critical and non-critical ones, if needed. Note that the only possible read-write conflict between active components occurs here, between the critical section of an active component and the sync transition of its active parent.

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 that is the target of a special transition that is fired when the event occurs. The behavior 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 (Figure 7).
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 8 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 its own start and end control places, through which it will acquire a token when it is scheduled.

### 4.2.2 Component Model-V as a Petri Net

So far we have shown how the different elements in the model can be mapped to Petri Nets representations. But we need to put all this information together and see how composite components schedule their parts.

Considering the case study presented in [NAD+02], we know that **FQD** is an event component, **ProcessApplication** is a passive component and **ModBus** is an active component, and their composition will be modelled as a composite component.

FQD has “out” ports **actual position** 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.

In a composite component, data ports can be connected either to represent the 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 Nets, there is no distinction – in both cases, the connected ports represent the same variable and are therefore modelled by the same, coalesced data place. This can be seen in Figure 9 where 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 that 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.

Figure 9 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 behavior (i.e. triggered by an event or running in a separate thread), the schedule is responsible only for synchronizing the data ports. The resulting net is clearly deadlock-free: the only conflicts between simultaneously enabled transitions occur where sync transitions compete with critical sections of active or event components. Since each of these transitions...
lock all the required data ports simultaneously, no \textit{waits-for} cycles are possible, and hence no deadlock can arise.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example_petri_net.png}
\caption{Example of a Petri Net model}
\end{figure}

5 Related Work

The field device component model is related on one hand to reactive and real-time languages, and on the other hand to component, component composition and architecture description. Hence, we present the related work regarding these axes. The information from this related work influenced the PECOS model only indirectly, so we do not draw direct parallels between the related work and the model.

5.1 General-Purpose Component Models

First of all we looked at general-purpose models that are used in the desktop computer world to find out whether they are (partially) applicable in the field device world. Various component models exist today, especially for general-purpose desktop systems. Their overall goals are similar to the ones of PECOS, namely more reuse, reduced development time and higher quality. As described by Szypersky in [Szyp98], component software engineering now promises to ‘deliver reusable, off-the-shelf software components for incorporation into large applications’. The basic idea of components is that of black-box reuse applied on wrapped binary components. Black-box reuse is considered better for reuse because it completely hides the internals, and to reuse the component people do not need to know those internals to compose components. So, components are typically defined as entities that encapsulate some internal representation with one or more interfaces. Other components can invoke functionality through these interfaces. From the reuse point of view, this provides some advantages:
It allows binary composition of components, something for which the white-box (source code) reuse of object-oriented programming has no language support.

Components can be implemented in different implementation languages (as long as they share a compatible interface and runtime mechanism) [Sieg96] [Mons00].

It becomes also easier to support distribution of components and develop client/server applications in an easier and more transparent way [Sieg96] [Mons00].

Some other languages focus on runtime (re)configuration of applications, allowing components to be added, removed or changed and even to change the overall architecture of the application at runtime [SG96].

What all these solutions have in common is that they need some runtime infrastructure to work. For example, to support distribution, centralized or distributed naming services have to be used. When bridging languages, at least some type-conversions are needed at runtime and certain calling strategies for methods/services have to be followed. Runtime configuration needs a complete and complicated runtime infrastructure.

As is explained in Section 2, the context of field devices at this moment does not allow for such runtime infrastructure. Instead, regarding the runtime, we are interested in deployment on single-processor devices with limited power and memory, where timing and scheduling are important. Hence the component model discussed in section 7 focuses on a static approach of composing the components, with runtime environment that focuses on the synchronous passing of data in a blackboard-like architecture. Hence most techniques applicable in general-purpose component models cannot be used in the context of field devices in the scope of this project. What we reused was some of the conceptual ideas:

1. the behavior of components is black-box, and completely encapsulated in the components
2. the model is language-independent, even if at this moment we do not consider composition of components in different implementation languages (because of the runtime cost this implies),
3. the separation between the implementation of a component, its interface, and the interconnection of components. This is advocated in all component models, and even in the software architecture world.

5.2 Architectural Description Languages

Since PECOS needs a component model that deals with communicating components, we looked at Architectural Description Languages (ADLs), since some of those target exactly that: communicating architectural components. In general, ADLs have been proposed as a notation to support architecture-based development, formal modeling, and analysis and development tools that operate on architectural specifications. An ADL is a language that provides features for modelling a software system’s conceptual architecture. ADLs provide a conceptual framework and a concrete syntax for characterising architectures [GMW97]. The building blocks of an architectural description are: components, connectors, and architectural configurations (or topologies). An ADL typically embodies a formal semantic theory. That theory is part of an ADL’s underlying framework for characterizing architectures; it influences the ADL’s suitability for modeling particular kinds of systems (e.g., highly concurrent systems) or particular aspects of a given system (e.g., static properties). Below we list the most widely used ADLs currently in use.

- Darwin/Regis [MDK94]: this environment focuses on supporting distributed applications. Components are single-threaded active objects and bindings represent communication links between them. It is tightly integrated with the ADL Darwin, and we are looking whether this could be applicable in the context of PECOS.
- Wright [Alle97] and Aesop [Garl94]: The goal of Wright is to provide a precise, abstract, meaning for an architectural specification and to analyse both the architecture of individual software systems and of families of systems. Wright also serves as a vehicle for exploration of the nature of the architectural abstractions themselves. The underlying model of Wright is CSP, which is the natural choice since it is focussed so much on connectors and connector styles. Experiments we did using Wright to represent an object-oriented framework in terms of software architectures [Arev00] showed that it indeed focuses heavily on connections between components that are running concurrently. Aesop is a toolkit for rapidly producing software architecture design and analysis environments that are customized to support specific architectural styles.
C2 [MOT97], SADL and Argo: C2 is a component- and message-based architectural style that sees an architecture as a hierarchical network of concurrent components linked together by connectors in accordance with a set of style rules. SADL is the ADL built to support C2 and, like Wright, focusses on concurrency aspects. Argo is the graphical design environment for constructing, analyzing, and generating C2 architectures.

Rapide [Kenn95]: The Rapide Language effort focuses on developing a new technology for building large-scale, distributed multi-language systems. This technology is based upon a new generation of computer languages, called Executable Architecture Definition Languages (EADLs) and a toolset supporting the use of EADLs in evolutionary development and rigorous analysis of large-scale systems.

ControlH and MetaH: ControlH is a language based on block diagrams and is especially useful for capturing a type of functional decomposition, especially in control engineering and signal processing and other non-software disciplines. Conceptually, blocks can be seen as transforming continuous time-varying input signals to continuous time-varying output signals. ControlH is primarily a functional or signal flow language rather than a procedural one. MetaH allows a specification of system components and connections, and attributes of those components and connections that are relevant to the real-time, fault-tolerant, secure partitioning, and multi-processor aspects of an application. ControlH seems to be partially useful in the context of PECOS, but not to express a complete device. MetaH seems to have some good ideas that we can use as well. The problem with both is that we have access only to descriptions on webpages, since they are commercial systems produced by Honeywell. We have contacted Honeywell for more information and possibly evaluation software.

ACME [GMW97] is not an ADL but an interchange format to be used between ADLs. As we said previously, ADLs must provide components, connectors and architectural configurations. Medvidovic and Taylor in [MT97] have developed a framework for classifying and comparing ADLs. In this work, they enumerate several aspects of both components and connectors that are desirable, but not essential: interfaces (for the connectors) and types, semantics, constraints, and evolution. Medvidovic and Taylor list desirable features of configurations such as understandability, heterogeneity, compositionality, constraints, refinement, traceability, scalability, evolution and dynamism.

5.3 Real-time Modeling Languages

Besides ADLs, we also looked at more classic, well-known model languages that deal with real-time modeling: Real-time Objected Oriented Modeling (Room) [SGW94] and UML-Realtime [SR98], an extension of UML based on Room to deal with real-time systems. Both introduce similar structural elements to model real-time systems:

- Capsules/Actors: A capsule models a complex, physical, possibly distributed architectural object that interacts with its surroundings solely through ports. A capsule may contain one or more subcapsules joined together with connectors. The internals of a capsule are described by a collaboration diagram.
- Ports: A port allows the communication between components (capsules/actors). Components communication is signal-based.
- Connectors: A connector is an abstract view of signal-based communication channels that interconnects two or more ports.

In the context of PECOS, UML-realtime could be used. However, some points are worth to be mentioned. First UML-Realtime is mainly a notation whose semantics is not clear. Then there is no description of behavior of the capsules regarding concurrency except the fact that the communication is signal-based. We chose not to use UML-Realtime and having to extend it with PECOS specific stereotypes.

5.4 Synchronous Languages

Beside modeling languages that still need to be mapped to an implementation, there also exist domain-specific languages that explicitly deal with timing and synchronization in the programming language. Synchronous languages like Esterel [BRS93], [Berr00], Lustre [HCRP91] and StateCharts [Hare87] [Hare88], were introduced in the 80s to program reactive systems. Such systems are characterised by their continuous reaction to their environment, at a speed determined by the latter.
Esterel is a synchronous and imperative concurrent language specifically dedicated to control-dominated reactive programs which are found in real-time process control, embedded systems, supervision of complex systems, communication protocols and HMI. In Esterel, programs are abstractions that manipulate input signals and generate output signals. [Berr00]. Once programs are expressed in Esterel they can be formally proved (i.e., non-reachability of state, timing constraints), compiled to C in a compact form, and also simulated. Automated test generation is also provided. Esterel supports the expression of components in terms of modules with generic input/output parametric types.

Tool support exists that allow the graphical but formal specification of programs, their interactive simulations, verification, the automatic test generation, and automated document generation. In the context of field devices and the specific requirements listed in Section 6, Esterel seems particularly interesting because the size of the generated code is suitable for field devices and, more important, the timing issues and memory consumption can be verified.

1. Esterel allows the verification that the output of a program is produced in a certain number of cycles after the input. This means that component time-substitutability could be verified.
2. Esterel allows different code generation schema: the first one is “boolean generation”. By counting the number of instructions we can then deduce exactly the size of a component and its exact execution time. The second is condition-based and can provide maximum execution time (worse case execution time) for a component [Ardi01].

While Esterel seems to match the requirements of field device components very well, we cannot because not all the component behavior is represented by a state automata and because we do not have access to the state automata that each component represents. Being able to express field device state automata in Esterel is definitely important future work.

SyncCharts [Andr96] is a graphical formalism dedicated to reactive System Modeling. Many features are inherited from StateCharts [Hare87], [Hare88] and Argos [MJLR94], [JMO93], [Mara91]. SyncCharts allows the specification of reactive behavior, as well as the synchronous programming of applications. Any syncChart can be automatically translated into an Esterel program. Argos is inspired by Harel’s statecharts. It offers both a graphical and a textual syntax. The main differences from statecharts are the use of a truly hierarchical composition operator and the application of a strict synchrony assumption. Argos is the basis of the programming environment Argonaute which provides a compiler and allows a lot of connections to verification tools.

Lustre is a synchronous declarative language for programming reactive systems [HCRP91]. It is declarative; a system description is a set of equations that must always be satisfied by the program variables. This approach was inspired by formalisms familiar to control engineers, like systems of differential equations or synchronous operator networks. A program variable in LUSTRE is considered to be a function of multiform time: it has an associated clock which defines the sequence of instants where the variable takes its values. However, Esterel seems to be more appropriate to describe components.

### 5.5 Piccola

Last but no least we looked at a language that is targeted at making composition of components explicit (and hence to support composition-based programming). Piccola is an experimental language for composing applications from software components [ALSN01] [AN01]. Piccola is defined by a thin layer of syntactic sugar on top of a semantic core based on Milner’s pi calculus. (Piccola stands for PI Calculus based COmposition LAnguage.). Piccola is designed to make it easy to define high-level connectors for composing and coordinating software components written in other languages. It goes beyond scripting languages because it is not biased towards one particular scripting paradigm. Instead, it allows components to be composed according to different compositional styles. As such, Piccola focuses on providing mechanisms for building different kinds of compositional abstractions, namely wrappers, adapters, connectors, coordination abstractions, and generic glue code.

In Piccola, everything is a “form”, a kind of immutable, extensible record that is useful for modeling objects, components, configurations, communications, default values and namespaces. Currently scripts exist that define and use different compositional styles such as aggregation, functional composition, inheritance, mixin composition, and stream composition.

However, Piccola is not well-suited to describe the composition of field device components because it is designed to express different composition styles in the context of heavily concurrent and distributed components.
6 Conclusion

This technical report describes the component model used to model field devices in the PECOS project. It specifies the structure and semantics of components, and a formalization with Petri nets. This forms the foundation for the tools and services needed to make the model usable in practice.

7 Reference


