Deliverable D.2.2.8-5

(previously D2.2.1 and D.2.2.2)

Field-Device Component Model-V

1 Identification

Project Id:	IST-1999-20398 PECOS
Deliverable Id:	D2.2.8-5 Component Model-V for Field Devices (previously D2.2.1 - D2.2.2. Meta Model for Component Specification and Meta Model for Architecture Specification)
Date for delivery:	2001-08-10
Planned date for delivery:	
Classification	Public
WP(s) contributing to:	WP 2
Author(s):	Gabriela Arévalo, UNIBE Stéphane Ducasse, UNIBE Oscar Nierstrasz, UNIBE Roel Wuyts, UNIBE

1.1 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 document 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 (D2.2.9) and the Pecos demo 3 (D1.4). The model enables composition rules to be introduced. Synchronisation and scheduling issues are partially specified. Thread priorities and task switching will be addressed in D2.2.6.

1.2 Keywords

Meta-model, component, composition, software architecture, synchronisation, component execution

1.3 Version history

Since this document supersedes both old deliverables D2.2.1 and D2.2.2, we have added the version history for these documents. Moreover, we have also indicated which iteration of the model these documents describe. Note that, starting from this document, explicit documents for each iteration will be delivered. Hence documents D2.2.8-X will be *living* documents, where the X indicates the iteration number of the model described in that document.

Ver	*Date*	*Editor(s)*	*Status & Notes*
1.0	28-01-01	Gabriela Arévalo	D2.2.1 - First version (iteration 1)
1.1	01-02-01	Roel Wuyts,	D2.2.1 - Second version (iteration 1)
1.2	02.02.01	Stéphane Ducasse,	D2.2.1 - Third version (iteration 1)
1.3	12.03.01	Roel Wuyts	D2.2.1 - (iteration 2). Comments from ABB included
2.0	12/10/01	Roel Wuyts	D2.2.2 - First version (iteration 2)
2.1	12/10/01	Gabriela Arévalo	D2.2.2 - Second Version (iteration 2)
2.2	25/05/01	Stéphane Ducasse	D2.2.2 - Third Version (final draft) (iteration 2)
2.3	06/06/01	Wuyts	D2.2.2 – Comments included (iteration 2)
4.0	08/08/01	Wuyts, Ducasse	D2.2.8 - Iteration 4. Complete new version with Prolog
5.0	12/09/01	Wuyts	D2.2.8 – Rewrite. Uses no Prolog anymore (iteration 4)
5.1	17/09/01	Wuyts	D2.2.8 – Draft before Amsterdam meeting (iteration 4)
5.2	25/09/01	Wuyts, Ducasse	D2.2.8-5 – Draft (iteration 5)
5.3	3/10/01	Ducasse	D2.2.8-5 – Draft (iteration 5) first feedback reinier
5.4	9/10/01	Ducasse	D2.2.8-5 – Draft (iteration 5) second feedback reinier
5.5	10/10/01	Nierstrasz, Ducasse	D2.2.8-5 – Draft (iteration 5) feedback reinier, peter
5.6	10/10/01	Ducasse	D2.2.8-5 – Draft (iteration 5) last feedback peter

1.4 Classification

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

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

1.5 Disclaimer

The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. Readers may use the information at their sole risk and liability.

2 Table of Contents

1 Identification 1
1.1 Abstract 1
1.2 Keywords 2
1.3 Version history 2
1.4 Classification 2
1.5 Disclaimer 2
2 Table of Contents 3
3 List of Achievments 4
4 Abstract 4
5 Introduction 4
6 Specific Requirements 4
7 Field-Device Component Model-V 6
7.1 Model Overview 6
7.2 Definitions of the model entities 7
7.3 Parent and Scope chain 8
7.4 Execution Model 9
7.4.1 Component Communication Synchronisation 9
7.5 Example 10
7.6 Field Device 12
7.7 Discussion 13
7.8 Open Issues 13
8 Related work 14
8.1 General-Purpose Component Models 14
8.2 Architectural Description Languages 14
8.3 Real-time Modelling Languages 15
8.4 Synchronous Languages 16
8.5 Piccola 16
9 Future Work 17
9.1 Domain-specific implementation languages 17
9.1.1 CeCil 17
9.1.2 AstLog 17
9.1.3 CoffeeStrainer 18
9.2 Formalization 19
10 References 19

3 List of Achievments

Report Title / Document Id	Contents

4 Abstract

The domain of embedded devices and specifically the domain of field devices urgently need 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 document 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 (D2.2.9) and the Pecos demo 3 (D1.4). The model enables composition rules to be introduced. Synchronisation and scheduling issues are partially specified. Thread priorities and task switching will be addressed in D2.2.6.

5 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 focussed instead on desktop applications. For example, operating systems or development environments are hard to find for embedded systems. The goal of the PECOS (PErvasive COmponent Systems) project is to apply solutions for component oriented development (COD) 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 6 of this document we present and motivate the specific requirements the field device component model is intended to address. In section 7 we informally present the structural and execution aspects of the model, and we illustrate the model by means of examples. Section 8 presents prior and related work.

6 Specific Requirements

Field devices are embedded reactive systems. A field device can analyse 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 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 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 40 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 of space requirements they typically offer only basic scheduling. However RT-OSs have a lot of 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 initialised) 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 analysing 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 scenario's 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 *synchronisation* and *concurrency*. Since solutions favoured 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 synchronisation 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 developer need not to be concerned with synchronisation or locking issues; the generated code needs to provide enough functionality in the form of an API so that the necessary data modelled 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 deliverables (D2.2.7 and D2.2.6 respectively).

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

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

7 Field-Device Component Model-V

This section starts with an overview of the model, then presents the model from a structural point of view and from an execution point of view.

7.1 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 synchronisation 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 three *property bundles* (*scheduling*, *memory* and *initialisation*), 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 four different kinds of components: *passive*, *active*, *event and timer*.

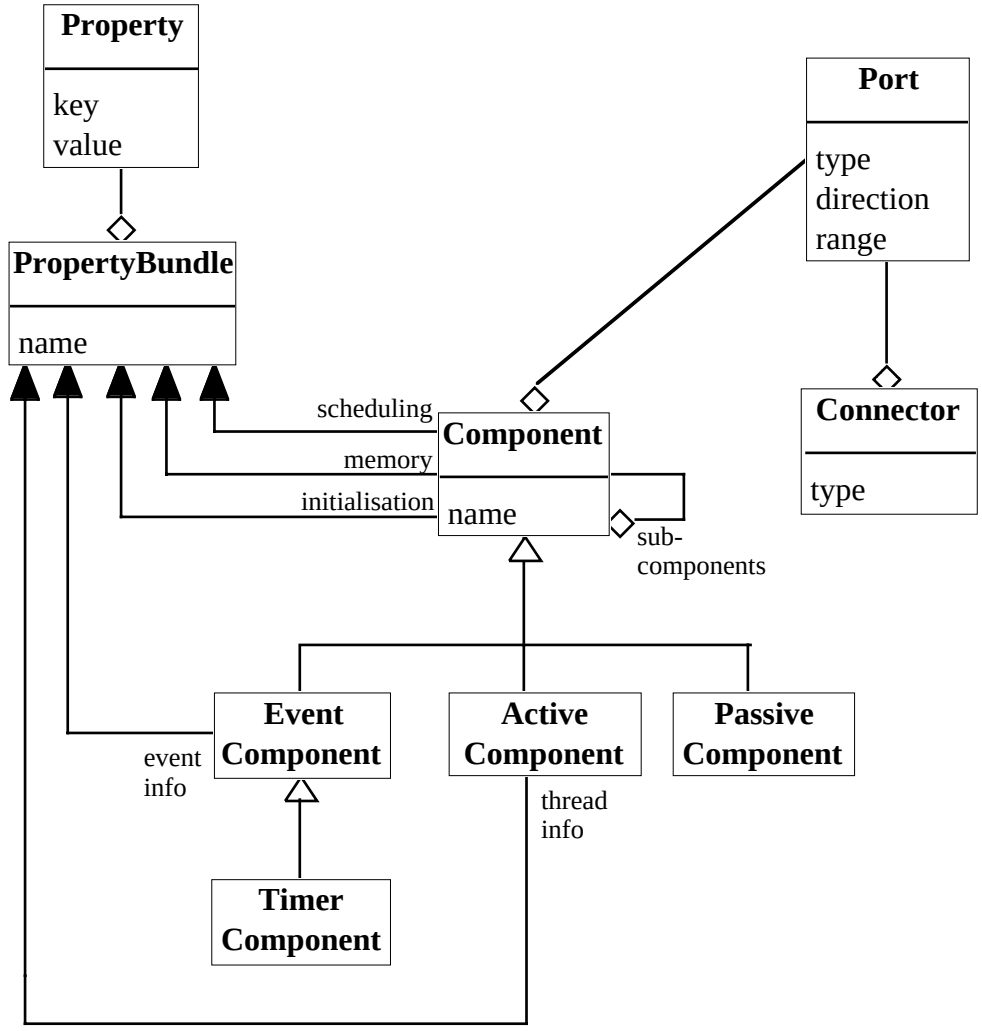

Figure 1: Pecos Component Model Diagram

7.2 Definitions of the model entities

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

Every component has a *name*, a number of *property bundles* and *ports*, and a *behaviour*. The *behaviour* of a component can be seen as a function or an algorithm that takes data available on the component ports or represented by some internal component data and produces some data on the component ports or side effects for example when the component wraps a physical device.

The *ports* of a component can be connected to ports of other peer components and internal operations. The internal operation can be behaviour specified in the form of an internal function, a composition of components, or both. Ports offer the sole mechanism for a component to interact with the outside world.

A *composite component* is a component that contains a number of connected sub-components. A composite component has *external ports* that are connected to selected ports of its sub-components. Note that the sub-components 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 7.6

a field device is modelled as a component hierarchy, i.e., a tree of component, with an *active composite component* at its root.

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 four 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 7.3). 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 used to model ongoing or longer-lived activities that do not complete in a short time-cycle.

- **Event Component**. An *event* component is an active component whose behaviour 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. Information about the event (including extra timing and memory information, such as the extra load put on the processor for handling events) is described in the event property bundle. Whenever the event *fires*, the behaviour is executed immediately (see below the execution model).

- **Timer Component**. A *timer* component is a special kind of event component that is used to model timers. The idea is that the timer can be set to go off at a certain time (and then execute an action, just like any other event component). For example, a component might need to set a timer to go off in 10 milliseconds and then read a certain value. The difference with regular event components lies in the setting of the timer, that has to be done by another component while for regular event components the event is triggered by the hardware encapsulated by the component. Hence, a timer component is an event component that has to be a sub-component of the component that sets it (and components can only set timers that are direct sub-components). Setting the timer has to be done in the implementation (and this does not show up in the model under form of a special port or event).

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 the transfer of data over a connector is considered *instantaneous* (see Section 7.4*)*. Moreover 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 characterise 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. The description of the components in the document D1.4 presents the property bundles that have been identified until now.

Note. Ports and connectors may have properties. This is an open question.

7.3 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 sub-components[1]. Every one of these sub-components can again contain passive or active sub-components. 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 sub-component *w*, that in turn contains a sub-component *x*, then the scope chain for *x* would be {*w*, *v*}, in this order.

We define the *active ancestor* of a component as to the first *active* component contained in the scope chain of this component.

7.4 Execution Model

The Section 7.3 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 synchronised 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 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 synchronised to maintain consistency.

To explain the execution model and to show how the components synchronise 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 synchronise 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.

7.4.1 Component Communication Synchronisation

The communication of parallel components requires a synchronisation 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 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, synchronisation is exercised by limiting the sub-tree access to its environment.

[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.

Passive component. While a passive component gets the token, its internal behaviour is run. 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 synchronised.

Active component. An active component runs concurrently. While an active component is running, it may control and schedule its own sub-components, but 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 synchronise its internal ports with its external ports. It can decide which data have to be synchronised 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 synchronised with the thread of its active ancestor. The details of this synchronisation 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 behaviour,
- 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 synchronisation 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 or Timer component. An event or timer 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.

About Data Transfer. Data transfer through the hierarchy over connected connectors has no delay unless active components are involved. Neither connectors nor ports on passive components have delays.

7.5 Example

In this section we give an example to illustrate the behaviour of a model. Note that we are again not considering how to calculate the schedules, but to show how a model works given a certain schedule.
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 (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.

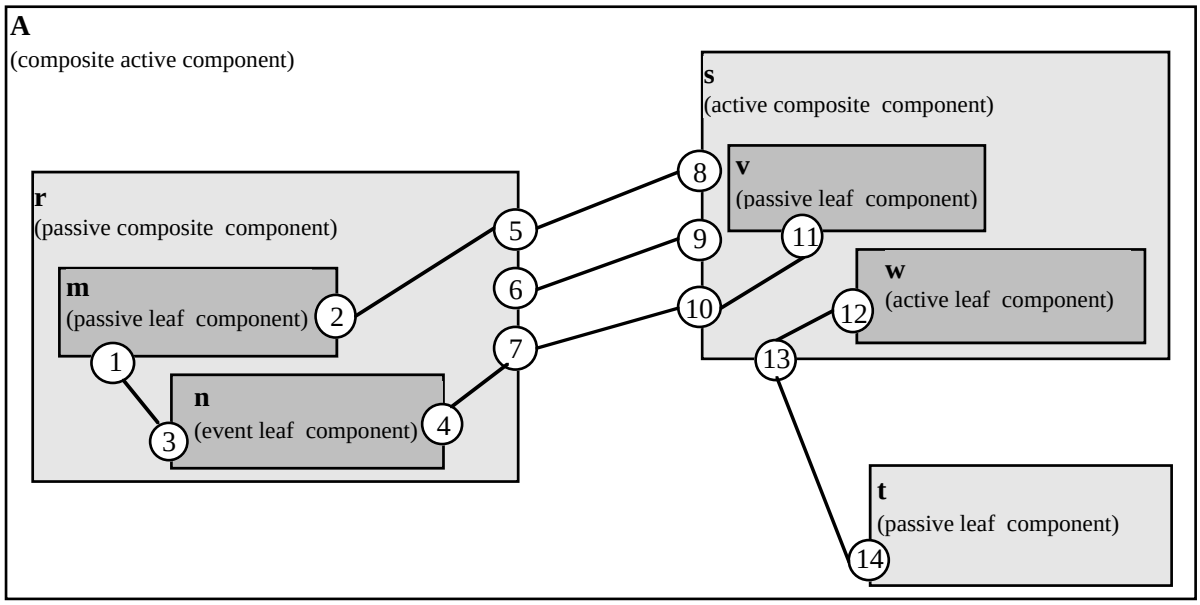

Figure 2: Example of a composite active component containing leaf and composite sub-components. Note that this is not a complete model (it omits some information for clarity).

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*[2].

[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 deliverable D2.2.6.

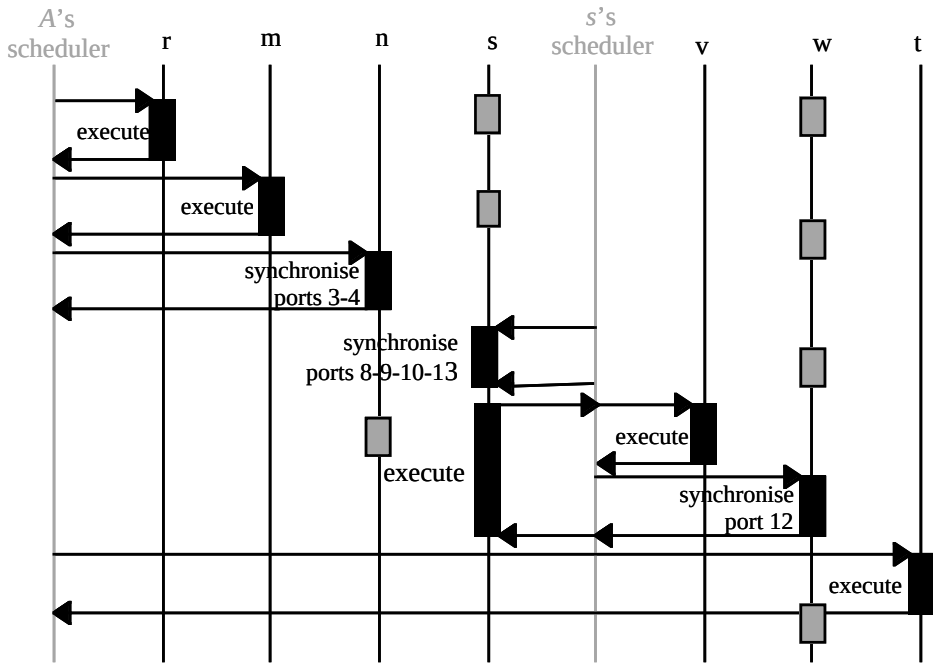

Figure 3: Component Sequence Diagram of a run of the model shown in Figure 2.

With these two schedules defined, a possible run can look like the diagram shown in Figure 3. This diagram shows the names of component instances in black, and the schedulers in grey. The arrows indicate the scheduling of components by the scheduler, i.e., when a component gets the token. Hence, every box on a line of a component indicates the time slot during which the component has the token. Hence at that moment the component acts as described in Section 7.5. The text besides the box contains a comment, such as what ports are synchronised. Whenever boxes are at the same height in the model, they are executed concurrently. The grey boxes represent the execution of active components which do not currently have access to their external ports.

Of course, this is just one possible run. However, we show some important points that might have escaped in the description of the behaviour

- The behaviour in active components can run whenever it likes. This is only possible because does not access external data ports, and hence it doesn't influence the overall component. It's only at the specified times in the schedule that the data from the external ports is synchronised with the rest of the component. Note that the implementation has to take care of synchronising the thread of the active component with that of its active ancestor. This will be described in detail in the documents dealing with code generation and the language mapping (deliverable D2.2.9).
- Likewise, event components can react on events, in the same way that the internal thread of an active component works: by only using the internal information and synchronising during the timeslot.

7.6 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 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 modelled, ports and connections that are common to all devices will be added.

7.7 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.

Synchronisation. 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 synchronised by the token mechanism. Of course, this does not come for free: they need to wait for the token to pass by.

7.8 Open Issues

The following issues are currently open:

- Task switching between concurrent threads is not explicitly modelled. This is needed to reason about timing constraints.
- The semantics of thread priorities is not defined. This is needed to reason about liveness (in particular, starvation and livelock).
- The synchronisation of active components with their active ancestors is not specified. In practice, several strategies can be used. The field device component model should specify the semantics of the synchronisation mechanism to enable reasoning about safety.
- The model has not yet been specified formally. A timed process calculus would likely offer the best setting to both formally specify the execution model and to enable reasoning about safety, liveness and timing constraints.

These issues will be addressed in the next iteration of this document.

8 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.

8.1 General-Purpose Component Models

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 5, 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 behaviour 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.

8.2 Architectural Description Languages

Architectural Description Languages (ADLs) have been proposed as a notation to support architecture-based development, formal modelling, 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 characterising architectures; it influences the ADL's suitability for modelling 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 understandibility, heterogeneity, compositionality, constraints, refinement, traceability, scalability, evolution and dynamism.

8.3 Real-time Modelling Languages

In this section we look at two well-known modelling languages for real-time systems: *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.

8.4 Synchronous Languages

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.

8.5 Piccola

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.

9 Future Work

9.1 Domain-specific implementation languages

In the current setup, developers need to implement the code for the component in a general-purpose programming language. In the context of the project this can be C++ or Java. However, this poses problems regarding safety, because either languages contain facilities that should not be used. For example, the context of field devices does not allow to allocate memory after initialisation time. So, developers that write code shouldn't be allowed to do that. Depending on the exact choices that will be taken in the skeleton code generation, there will be other constraints on the code that developers can write. Instead of leaving these constraints implicit, we could express them in constraint languages, and hence give developers a domain-specific language. The advantages of this approach are two-fold:

- developers do not have to learn a new language, as it's a constrained language they already know
- the programming conventions are made explicit and can be checked

Since the languages used in the PECOS Project are C++ and Java, we immediately see the following possible candidate languages to express the constraints: CeCil and AstLog for C++ and CoffeeStrainer for Java.

9.1.1 CeCil

CCEL [MDR93] allows us to express and enforce constraints on C++ code, such as *The member function M in class C must be redefined in all classes derived from C*, *If a class declares a pointer member, it must also declare an assignment operator and a copy constructor*, or *All classnames must begin with an upper case letter*. The constraints are included in the source files in specially formatted comments. Syntactically, CCEL constraints look like expressions in first-order predicate calculus, allowing programmers to make assertions involving existentially or universally quantified CCEL variables. Constraints can be grouped in constraint classes, but there are no provisions for composing such classes, or calling constraints from one class in another class. As example we show a CCEL constraint class, with two constraints that express that whenever a C++ class contains a pointer member, or inherits from a class containing a pointer member, it must declare an assignment operator:

```
PointersAndAssignment {
      // If a class contains a pointer member,
      //  it must declare an assignment operator:
      AssignmentMustBeDeclaredCond1 (
          Class C;
          DataMember C::cmv | cmv.is_pointer();
          Assert(MemberFunction C::cmf; | cmf.name() == "operator=");
      );
      // If a class inherits from a class containing a pointer member,
      // the derived class must declare an assignment operator:
      AssignmentMustBeDeclaredCond2 (
          Class B;
          Class D | D.is_descendant(B);
          DataMember B::bmv | bmv.is_pointer();
          Assert(MemberFunction D::dmf; | dmf.name() == "operator=");
      );
 };
```

9.1.2 AstLog

Another constraint system that works on C++ code is Astlog [Crew97]. Astlog is a logic programming language with two specific additions that facilitate the reasoning over parse trees. First it avoids the overhead of translating the source code into the form of a Prolog database by allowing predicates to work directly on C++ code. Second, terms are matched against an implicit current object, rather than simply proven against a database of facts, leading to a distinct 'inside-out functional' programming style. Using Astlog we can perform logic queries on C++ code to compare the structure of two parse trees. Hence we can use it to check whether the C++ implementation conforms to the structure we describe in the query (but not to generate code, for example). The result of the query is the report that indicates whether, and where, the implementation conforms to the design. As example we show an Astlog predicate *sametree* that is used to compare two parse trees (the current node that is implicit, and the passed argument, *node*). The predicate holds if root nodes have the same opcode and all corresponding children have the same structure:

```
sametree(node)
      <-  op(nodeop),
          with(node, op(nodeop)),
          not(and(  with(node, kid(n, nkid)),
                    kid(n, not(sametree(nkid))))));
```

9.1.3 CoffeeStrainer

CoffeeStrainer [Boko99] is a system that allows us to statically check structural constraints on Java programs. The constraints are written in stylised Java, and have access to the complete Java parse tree. The scope of the constraint is determined by its position in the source code. For example, a constraint that appears in a class or interface applies to that class or interface and its subtypes. Subclasses can strengthen constraints, but never weaken them. As an example we show a CoffeeStrainer constraint that specifies that subclasses that override a method *initialize* should first call *super.initialize()* before doing anything else:

```
public abstract class MediaStream {
    public void initialize() {
    /*-
        private AMethod initializeMethod() {
            return Naming.getInstanceMethod(thisClass,
                "initialize", new AType[0]);
        }
        private boolean overrides(AMethod m1, AMethod m2) {
            if (m1 == null) return false;
            if (m1.getOverriddenMethod() == m2) return true;
            else return overrides(m1.getOverriddenMethod(), m2);
        }
        private AStatement getFirstStatement(ConcreteMethod m) {
            AStatementList s1 = m.getBody().getStatements();
            return s1.size() >0 ? s1.get(0) : null;
        }
        private boolean callsInitialize(AStatement s) {
            if(!(s instanceOf ExpressionStatement)) return false;
            AExpression e=((ExpressionStatement) s).getExpression();
            if(!(e instanceOf InstanceMethodCall)) return false;
            AMethod called=(InstanceMethodCall e).getCalledMethod();
            return called == initializeMethod()
                || overrides(called, initializeMethod);
        }
        public boolean checkConcreteMethod(ConcreteMethod m) {
            rationale = "when overriding initialize, " +
                "super.initialize() must be the first statement";
            return implies(overrides(m, initializeMethod()),
                callsInitialize(getFirstStatement(m)));
        }
    -*/
```

9.2 Formalization

The context of embedded devices is a very specific one. We already told, for example, that memory allocation is not possible outside the initialisation phase. As a result, formalized approaches that might not be applicable to general-purpose component systems might be applicable on component systems for embedded devices. This could help in supporting the checking of non-functional requirements.
We also believe that an important aspect of formalization is to represent the behaviour of components with state automata. This opens up possibilities to use several existing formalisms that we discussed in the Related Work chapter, such as Esterel, timed state automata, or ControlH/MetaH.

10 References

[Andr96] C. André. Representation and Analysis of Reactive Behaviors: A Synchronous Approach CESA'96, IEEE-SMC, Lille(F), July 9-12, 1996.

[Ardi01] L. Arditi, R&D Texas Instrument France, personal communication about ESTEREL and proof and validation of DSP and embedded real-time devices, 2001.

[ALSN01] F. Achermann, M. Lumpe, J.-G. Schneider, and O. Nierstrasz. Piccola – a small composition language. In H. Bowman and J. Derrick., editors, Formal Methods for Distributed Processing, an Object Oriented Approach. Cambridge University Press., 2001. to appear.

[AN01] F. Achermann and O. Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In M. Aksit, editor, Software Architectures and Component Technology. Kluwer, 2001. to appear.

[Alle97] R. Allen. A Formal Approach to Software Architecture. Ph. D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997

[Arev00] G. Arévalo. Architectural Description of Object Oriented Frameworks. Master Thesis. Ecole des Mines de Nantes and Vrije Universiteit Brussels. August 2000.

[BRS93] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In ACM, editor, Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Charleston, South Carolina, January 10–13, 1993, pages 85–98. ACM Press, 1993.

[Berr00] G. Berry. The foundations of Esterel. MIT Press, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.

[Boko99] B. Bokowski. Coffeestrainer: Statically-checked constraints on the definition and use of types in java. In Proceedings of ESEC/FSE'99. Springer-Verlag, September 1999.

[CES71] E.J. Coffman, M. J. Elphick, and A. Shoshani, System deadlocks. ACM Computing Survey 3,2 (June), 67-78, 1971.

[Crew97] R. F. Crew. Astlog: A language for examining abstract syntax trees. In Proceedings of the USENIX Conference on Domain-Specific Languages, 1997.

[Garl95] D. Garlan. What is Style ? In Proc. First International Workshop Software Architecture, April 1995

[Garl94] D. Garlan, R. Allen, and J. Ockerbloom, Exploiting Style in Architectural Design Environments, Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineerng, December 1994.

[GMW97] D. Garlan, R. Monroe, and D. Wile: ACME, An Architecture Description Interchange Language, White-paper, 1997

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language Lustre. In Proceedings of the IEEE, September 1991.

[Hare87] D. Harel, StateCharts: A Visual Formalism for Complex Systems, Science Computer Programming, 8, 1-12, 1987.

[Hare88] D. Harel, On Visual Formalisms, *CACM*, vol. 31, no. 5, May 1988, pp. 514-530.

[JMO 93] M. Jourdan, F. Maraninchi, and A. Oliveiro, Verifying quantitative real-time properties of synchronous programs, *Proceedings of Conference on Computer Aided Verification (CAV), Elounda, Greece, 1993*.

[Kenn95] J. J. Kenney, Executable Formal Models of Distributed Transaction Systems based on Event Processing, . Ph.D. Thesis, Stanford University, December 1995.

[MDR93] S. Meyers, C. K. Duby, and S. P. Reiss. Constraining the structure and style of object-oriented programs. Technical Report CS-93-12, Department of Computer Science, Brown University, Box 1910, Providence, RI 02912, Apr. 1993.

[MJLR94] F. Maraninchi, M. Jourdan, F. Lagnier and P. Raymond. A multiparadigm language for reactive systems. In Proceedings of the IEEE Internal Conference on Computer Languages, 1994.

[MDK94] J. Magee, N. Dulay and J. Kramer, Regis: A Constructive Development Environment for Distributed Programs. In IEE/IOP/BCS Distributed Systems Engineering, 1(5), pp. 304-312, September 1994

[Mara91] F. Maraninchi. The argos language: Graphical representation of automata and description of reactivesystems. In Proceedings of the IEEE Internal Conference on Visual Languages, 1991.

[MOT97] N. Medvidovic, P. Oreizy, and R. N. Taylor, Reuse of Off-the-Shelf Components in C2-Style Architectures. In Proceedings of the 1997 Symposium on Software Reusability (SSR'97), pages 190-198, Boston, MA, May 17-19, 1997. Also in Proceedings of the 1997 International Conference on Software Engineering (ICSE'97), Boston, MA, May 17-23, 1997.pages 692-700.

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

[MT97] N. Medvidovic and R. Taylor. A framework for classifying and comparing architecture description languages. In Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97), pp. 60-76, Lecture Notes in Computer Science Nr. 1013, Springer-Verlag, September 1997

[RJFC94] F. C. Ribeiro Justo and P. R. Freire Cunha. Deadlock-free configuration programming. In Proceedings of the 2nd International Workshop on Configurable Distributed Systems, March 1994.

[SGW94] B. Selic, G. Gullekson and P. T. Ward, Real-Time Object-Oriented Modeling, John Wiley & Sons, 1994.

[SR98] B. Selic and J. Rumbaugh, Using UML for Modeling Complex Real-Time Systems, Rational Rose White Paper, 1998.

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

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

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

[Wuy98] R. Wuyts. Declarative reasoning about the structure object-oriented systems. In Proceedings of the TOOLS USA '98 Conference, pages 112–124. IEEE Computer Society Press, 1998.