PECOS in a Nutshell

Thomas Genssler[1] Alexander Christoph[1] Benedikt Schulz[1]

Michael Winter[1] Chris M. Stich[2] Christian Zeidler[2] Peter Müller[2]

Andreas Stelter[2] Oscar Nierstrasz[3] Stéphane Ducasse[3] Gabriela Arévalo[3]

Roel Wuyts[3] Peng Liang[3] Bastiaan Schönhage[4]

Reinier van den Born[4]

September 16, 2002

[1] {genssler|christo|bschulz|winter}@fzi.de,
Forschungszentrum Informatik (FZI), Germany,
http://www.fzi.de

[2] {christian.stich|christian.zeidler|peter.o.mueller|andreas.stelter}@de.abb.com,
ABB Corporate Research, Germany,
http://www.abb.com

[3] {oscar|ducasse|arevalo|wuyts|liang.peng}@iam.unibe.ch,
Software Composition Group (SCG), University of Bern, Switzerland,
http://www.iam.uinbe.ch/~scg

[4] {Bastiaan_Schonhage|Reinier_van_den_Born}@oti.com,
Object Technology International (OTI), The Netherlands,
http://www.oti.com

Preface

The PECOS Project

Software is more and more becoming the major cost factor for embedded devices. In fact, software accounts for more than 50 percent of the development costs of such a device. The PECOS (PErvasive Component Systems) project seeks to overcome this by providing a component-based technology for the development of a specific class of embedded systems known as ”field devices”. It takes into account the specific properties of this application area.

Therefore the PECOS project has developed a component model, a composition language, and tools for field device software development which help to overcome the shortcomings of the current practice. The PECOS approach covers the whole software life-cycle of a field device. It defines a detailed software process which coordinates all development steps from requirements specification to deployment.

The PECOS project consortium consists of four partners: ABB Corporate Research Centre in Ladenburg, Research Centre for Information Technologies (FZI), the Software Composition Group at the University of Bern (SCG), and Object Technology International (OTI).

The roles of the partners within the project has been the following. ABB acted as main contractor, project coordinator and user in the project: ABB's Instruments Business Unit is developing a large number of different field devices and aims at introducing component based technology in their production. In addition, ABB has got expert knowledge in the field device domain and has carried out a number of case studies within the project. FZI and SCG have been charged with the research part of the project, which concerned the PECOS component model and the composition language CoCo. OTI is a tool provider, especially known for their products VisualAge for Java and Visual Age Micro Edition. They provided the knowledge of building development tools to support the PECOS methodology, e.g., concerning the Eclipse platform.

The PECOS project has been funded by the European Union (under the IST project number IST-1999-20398) and the State of Switzerland.

About This Handbook

This Handbook presents the PECOS approach for developing field device software. It has been written for the developer who wants to apply the method in a realistic setting. The main objective of this Handbook is thus to provide all the information that is necessary to understand and to apply the PECOS approach in practice.

In order to meet this objective, it does not only present the fundamental concepts and techniques, the PECOS approach is relying on, but also demonstrates how these principles are applied. For this purpose a detailed case study is presented which is used to illustrate and discuss each step of the PECOS software process. Thereby, the reader is provided with a number of practical recipes which answer all kinds of questions that emerge during the application of the PECOS method.

To understand this Handbook, no previous knowledge about component-based development is required. However, as the PECOS component model is mapped to C++ and Java, we assume

that the reader is familiar with one of these programming languages. Nevertheless, it should be possible to understand most of this Handbook without knowing these languages.

Besides, it has to be pointed out, that this Handbook could not describe the latest versions of the tools supporting the PECOS approach, as they will be available at the end of the project. This is due to the fact that the development of these tools proceeded in parallel to the writing of this book.

Handbook Structure

Chapter 1 gives a brief introduction to the domain of field devices and motivates the PECOS approach.

Chapter 2 contains a brief but practical introduction into PECOS. By means of a simple example, the chapter will provide the reader with some basic knowledge of the PECOS Run-Time Environment, the tool environment, the CoCo component language and PECOS schedules.

Chapter 3 gives an overview of the structural and dynamic aspects of the PECOS component model. It uses the example from Chapter 2 to illustrate the concepts and lighten the explanation.

Chapter 4 explains the CoCo language, which is an implementation of the PECOS component model introduced in chapter 3. It gives a brief overview of the CoCo language constructs and shows how CoCo is used to specify PECOS components and complete PECOS field devices.

Chapter 5 is concerned with the behaviour of components and the device as a whole. It starts by looking at the issues related to local component behaviour, giving hands-on advice on how to overcome problems. Then it goes on to show how PECOS devices are run, paying special attention to scheduling and timing.

Chapter 6 presents a detailled PECOS case–study, using C++. It explains, how the available PECOS tools are applied to model and implement a field device in a real–world setting.

Contents

1 Introduction **9**
1.1 What is an Embedded System, what is a Field Device? . . . 9
1.2 Current Field Device Development . . . 10
1.3 Motivation of the PECOS approach . . . 11

2 Babysteps with PECOS **13**
2.1 Introduction . . . 13
2.2 The CoCo Clock Example . . . 13
2.3 Setting up the Project . . . 14
2.4 My First CoCo . . . 16
2.5 Implementing the Behaviour . . . 17
2.6 Get the Clock Going . . . 18
2.7 Adding a Digital Display Component . . . 21
2.8 Update the Device to include the Digital Display Component . . . 22
2.9 Wrap up . . . 24

3 Component Model **25**
3.1 Introduction . . . 25
3.2 Structural Overview . . . 25
3.2.1 Components . . . 25
3.2.2 Ports . . . 26
3.2.3 Connectors . . . 27
3.2.4 Composite Components . . . 27
3.2.5 Properties and Properties Bundles . . . 27
3.2.6 Parent . . . 28
3.3 Execution Model . . . 28
3.3.1 Data spaces . . . 28
3.3.2 Execution and Synchronisation Behaviour . . . 29
3.3.3 Runtime Semantics . . . 29
3.4 Summary . . . 30

4 The CoCo Language **31**
4.1 Introduction . . . 31
4.2 Keywords . . . 31
4.3 Components . . . 31
4.4 Ports . . . 32
4.5 Composite Components . . . 33
4.5.1 Component Instances . . . 33
4.5.2 Connecting Component Instances by Connectors . . . 33
4.5.3 'Upward' Connectors . . . 33
4.5.4 Semantics of Connectors . . . 34
4.6 Properties . . . 35

4.6.1 Simple Properties 35
4.6.2 Property Bundles 37
4.7 Schedule Specification 37
4.8 Data Types 39
4.9 Summary 40

5 Component Behaviour **43**
5.1 Introduction 43
5.2 Application Structure 44
5.3 Component Structure 44
5.3.1 Component Type Base Classes 45
5.3.2 Component Base Classes 46
5.3.3 Component classes 48
5.4 Component Behaviour 49
5.4.1 Initialisation 49
5.4.2 Execution 50
5.4.3 Synchronisation 51
5.5 Scheduling Behaviour 53
5.5.1 Priority scheduling 56
5.6 Main and PecosDevice 57
5.7 Summary 58

6 Application building and deployment **59**
6.1 Application Structure 59
6.2 Component and Device Specification 60
6.3 Embedded Development Environment, M16C and embOS 64
6.4 Implementing Main 66
6.5 Implementing Component Behavior 67
6.6 Makefile and Building 67
6.7 Debugging and Deplyoing 68
6.8 Summary 68

A CoCo Grammer **71**

List of Figures

1.1 A simple fluid control system . . . 10

2.1 Model of the Clock Device example . . . 14
2.2 Create of a new PECOS Java Project . . . 14
2.3 Adding the PECOS RTE to the classpath . . . 15
2.4 PECOS Java Project Properties . . . 15
2.5 Creating a new PECOS component . . . 16
2.6 Outline view of a PECOS device . . . 17
2.7 Outline view of a PECOS device schedule . . . 18
2.8 Launching a PECOS application . . . 19
2.9 Output of a PECOS application . . . 20
2.10 Adding the SWT toolkit to the classpath . . . 21
2.11 Model of the extended Clock Device example . . . 22
2.12 Setting of VM arguments for SWT library . . . 23
2.13 Output of the PECOS Clock Device . . . 24

3.1 Model of the extended Clock Device example . . . 25
3.2 Tree view of the the Digital Clock example . . . 28
3.3 Possible Execution trace for the Digital Clock example . . . 29

4.1 Class Diagram of Tasks, Jobs and Activities . . . 38
4.2 Summary of CoCo's top–level syntax elements. . . . 41

5.1 Application Structure . . . 44
5.2 Classes for active component `Comp` . . . 45
5.3 Nested Components with their Class Diagram . . . 46

6.1 A simple valve positioner . . . 60
6.2 A very simple valve positioner. . . . 61
6.3 Required executing order and timing of our device . . . 61
6.4 Eclipse development environment with opened valve controller project . . . 63
6.5 Adding generated files . . . 66
6.6 TaskingWorkspace . . . 67
6.7 Make . . . 68
6.8 CrossView Debugger . . . 68

Chapter 1

Introduction

Since computer has become smaller, faster, more reliable and cheaper their range of applications has been widened. Built initially as equation solvers, their influence has extended into all areas of life. One of the fastest expanding areas is that of embedded real-time computers. It has been estimated that 99% of the worldwide production of microprocessors is used in embedded systems.

This tutorial is concerned with the development of a special class of embedded computers called field devices. The approach presented here was developed in the PECOS project (**P**ervasive **Co**mponent **S**ystems). By following this tutorial the reader will get a good understanding of PECOS and will learn how to use PECOS to develop software for embedded real-time systems.

This chapter provides you with an introduction into PECOS. It starts with an overview on the state of the art in today's software development for field devices. It presents the domain of embedded systems and especially field devices which have been used for the PECOS case studies. The main deficiencies of current embedded software development practice are presented and we show, how the PECOS approach tackles them.

1.1 What is an Embedded System, what is a Field Device?

Embedded Devices are not just small computers. Before starting, it is worth to define the phrase real-time systems. The Oxford Dictionary of Computing gives the following definition:

> Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness.

Timeliness must be taken from the context of the total system. Take a valve controller as an example. Its output is typically required within a few milliseconds, whereas for a temperature controller, the response may be required only within a second. A key feature is the role of the computer as an information processing component within a larger engineering system. It is for this reason that such applications has become known as embedded computer systems. Most of the critical properties of embedded systems are non-functional like real-time, fault recovery, low power consumption, security and robustness [1]. Further down we will see how PECOS especially supports real-time and robustness properties.

A field device is an embedded system often used in the area of process control. Field devices make use of sensors to continuously gather data, such as temperature, pressure or rate of flow. They analyse and reconcile this data, and react by controlling actuators, valves or motors. Field devices must provide high quality and reliability since malfunction may be dangerous or may involve high financial risk.

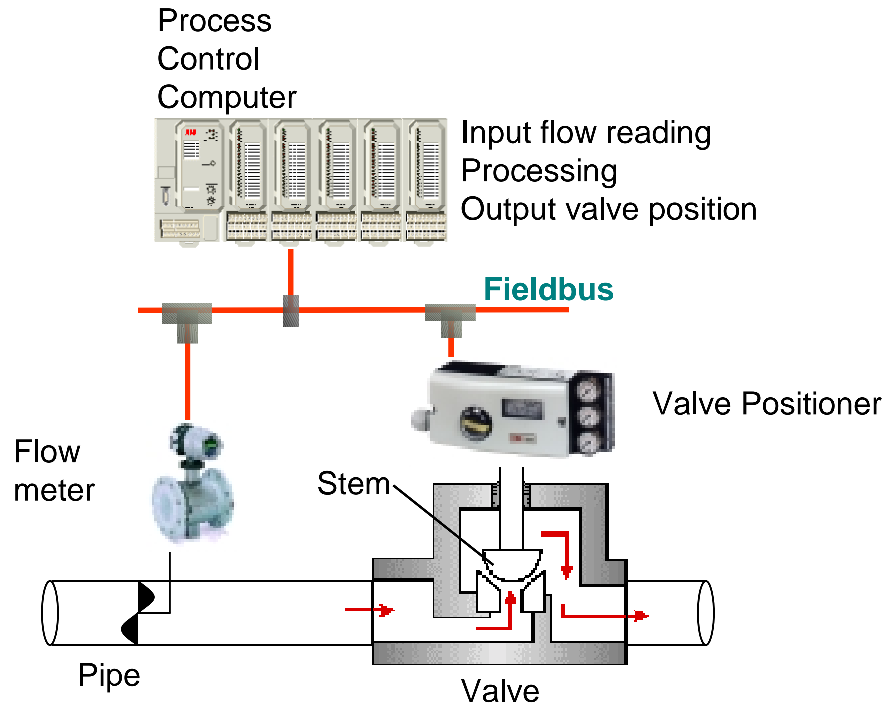

Figure 1.1: A simple fluid control system

Consider the simple example in figure 1.1. The process control computer performs a single activity: that of ensuring an even flow of liquid in a pipe by controlling a valve. On detecting an increase in flow by reading the current flow value from the flow meter, the computer must respond by altering the valve's stem position by writing a new valve position to the valve positioner. This response must occur within a finite period if the equipment at the receiving end of the pipe is not to become overloaded. Note that the actual response may involve quite a complex computation in order to calculate the new valve angle. Required response time may be in the range of seconds. Another example is the valve positioner itself. A valve positioner is a device used to increase or decrease the air pressure operating the actuator until the valve stem reaches the position called for by the process control computer. A new output signal must be generated e.g. every 20ms.

Often field devices are used in hazardous areas like chemical plants. The energy passing into the hazardous area is therefore limited in a way that regardless of the fault in the hazardous area, sufficient energy cannot be released to ignite an explosive atmosphere. As a consequence power consumption per device is limited. This has far reaching consequences on the design of field devices. Therefore typical field devices still use 8 or 16 bit micro-controllers (e.g. Mitsubishi M16C with 256k ROM and 20k on-chip RAM) today. This is especially because of their low power consumption. As a consequence (hardware and software) designs are often totally device specific to reach the low power requirements.

1.2 Current Field Device Development

How does a typical development look like? In many cases small groups are responsible for the whole development life-cycle from the product specification over design and development up to the users manual. Only one or two (senior) developers do understand the device in all its details. Others only may know a special part of the device like the control loop or the fieldbus commu-

nication interface. People often spend years in maintaining and tuning a developed product. Usually there is no separate group of people responsible to develop base technology for all devices developed at one location or even business unit wide (e.g. a competence team for a certain fieldbus protocol). Typically no other software development tools than an Integrated Development Environment (IDE) and an In Circuit Emulator (ICE) for real-time debugging are in use. Rarely one will find an up-to-date software development process or the use of design tools.

1.3 Motivation of the PECOS approach

As a consequence of the things said above many shortcomings exists in the area of embedded software development. The following list presents the most important ones.

The field device software is hard to maintain and to extend: Based on the monolithic design with individual architectures, interfaces and scheduling methods, today's field device software is hard to maintain and to extend. This is all the more important if code from consultants and external service providers should be integrated.

Duplicated functionality: ABB Instrumentation is located around the world. Due to the missing process and (tool) support for a corporate software reuse the same functionality is often implemented at different development locations in different ways for different field devices.

Slightly portable implementations: Most of today's field device software is developed for the specific combination of an electronic device and its physical environment. Additionally non standard real-time operating systems and low level micro-controller routines result in solutions which are often not portable to other environments.

Most field device projects fail to meet their schedules: The software development is often the reason for this delay! Software for new products is often developed from scratch with little reuse of well-tried architectures or components.

The effort for Integration and Regression Tests is too high: In today's field device implementations the control, state and data dependencies between units are hard to identify. This often leads to higher efforts for integration and regression tests than needed, after changing small parts of the system.

To address this shortcomings of the current software development process and the constantly increasing requirements, ABB's Business Unit Instrumentation have to find better solutions for a lot of key issues:

- Training of the current engineering staff to address the much higher demand for methods, tools and techniques.

- Organization and culture: For a corporate collaboration and a functioning reuse of completed jobs ABB has to provide the appropriate organization and culture. The point will become more and more important because the high pressure to reduce cost often comes along with staff reduction. Cooperation between different development teams will be a key for success in the future.

- Reuse of technical solutions: As it was already proven from other software domains component-based software engineering would bring a number of advantages to the embedded systems world too: faster development times; the ability to secure investments through re-use of existing, well tried components; the ability for domain experts to interactively compose embedded systems software and to adapt the software to specific customers needs.

- Iterative processes: The most difficult features have to be developed first in the earlier iterations and refined in later iterations - risks are effectively mitigated since there are now several integration periods to identify and resolve problems. Requirement, feature changes and additions must be handled as the project proceeds through subsequent iterations. The iterations allow the developers to inspect the results. This will be especially effective with domain specific tools and development environments. Since the project will have been tested and integrated several times, the probability of significant errors remaining is low. This means that the overall quality of the product should be enhanced.

PECOS is the major instrument to address the last two issues, reuse and iterative processes. The following chapters will present the PECOS approach which enables component-based software development for embedded systems, specifically for field devices. The project aims to solve the major technological deficiencies of current processes by developing a specific component model for the constraints of field devices. The CoCo language syntacticaly represents this component model. A coherent software development process is defined for embedded components and applications. A development and testing toolkit including editors, code browsers, code generators suppors target development. Schedule computation and testing is also supported. Most of these tools are integrated in the PECOS component environment. This handbook enables you to use PECOS to build your own component-based systems.

Chapter 2

Babysteps with PECOS

By Bastiaan Schönhage

2.1 Introduction

In this chapter you will get a head start in PECOS. Using a simple example, the basics of the CoCo composition language and the Eclipse-based tool support will be explained. In later chapters a more detailed discussion of the component model, the process and the tools will be provided. The goal of this chapter is to get you acquainted with the PECOS Integrated Tool Environment and give you at least an idea of what is possible with PECOS. Have fun ...

Structure After a short explanation of the *CoCo Clock* example device in Section 2.2, Section 2.3 explains how to set up a new PECOS Java project in Eclipse 2.0 with the PECOS extensions. Section 2.4 describes the use of the composition language CoCo to specify the components that are part of the example. In addition to this, the section briefly describes what files are generated by the code generator upon a build.

In the next part, Section 2.5 adds the behaviour to the components and Section 2.6 shows how a CoCo specification of a schedule can get the clock going.

Sections 2.7 and 2.8 show the power of the PECOS component-based approach by adding two additional components to the system to display the current time graphically. The sections explain how the device and schedule have to be changed to include the new components. Finally, Section 2.9 wraps up the chapter and presents a summary of the steps taken to build a simple PECOS device.

2.2 The CoCo Clock Example

The upcoming sections show how easy it is to build a simple application based on the PECOS component model. The example application used is a *Clock Device* that displays the current time of the day. In the first example, the output is text-based and updated every second. In a later step the example will be extended with additional components that show the time graphically.

The *Clock Device* that we are building in this chapter is extremely simple: print the time and update the output every second. In order to model this in PECOS, we only need three components: one to provide the time, one to display the time and one component that models the device and contains both the `Clock` and `Display` component.

Figure 2.1 gives a schematic overview of our example *Clock Device*. The `Clock` component on the left has one output port (`msecs`) that contains the current time in milliseconds. The `Display` component on the right-hand side is used to format the time and print it to standard output. In order to achieve its goal, it has an input port called `time`. The third component is the `Device`

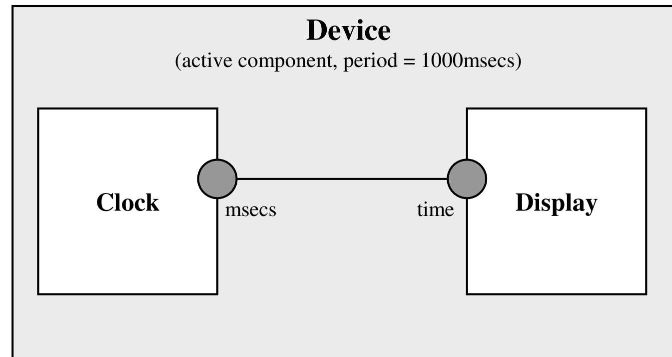

Figure 2.1: Model of the Clock Device example

component: it contains the clock and the display. The `Device` is an active component that runs its passive children every 1000 ms. This ensures us that the current time is displayed every second as long as the application is running.

2.3 Setting up the Project

Before we can actually develop the components and run the application, we first need to set up a project that contains our specifications and code. In order to achieve this, start up Eclipse (with the PECOS tools installed) and select the *New Wizard* from the menu `File-> New-> Other ...` (or press Ctrl-N).

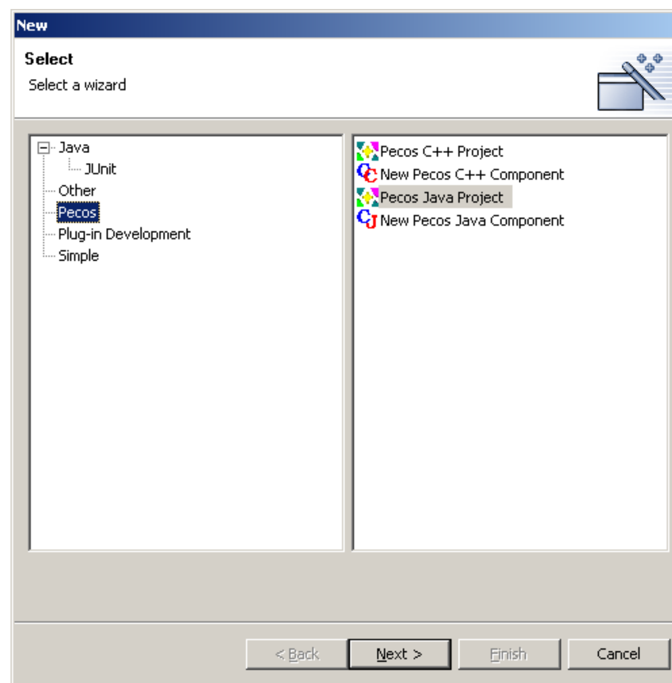

Figure 2.2: Create a new PECOS Java Project

In the project wizard (Figure 2.2) that pops up now, you should select `Pecos Java Project`. After pressing `Next` you should give the project an appropriate name; use `clock` for our example. In the final step of the wizard make sure that you add the Run-Time Environment to the

classpath of the newly created project using `Add External JAR ...` (see Figure 2.3).

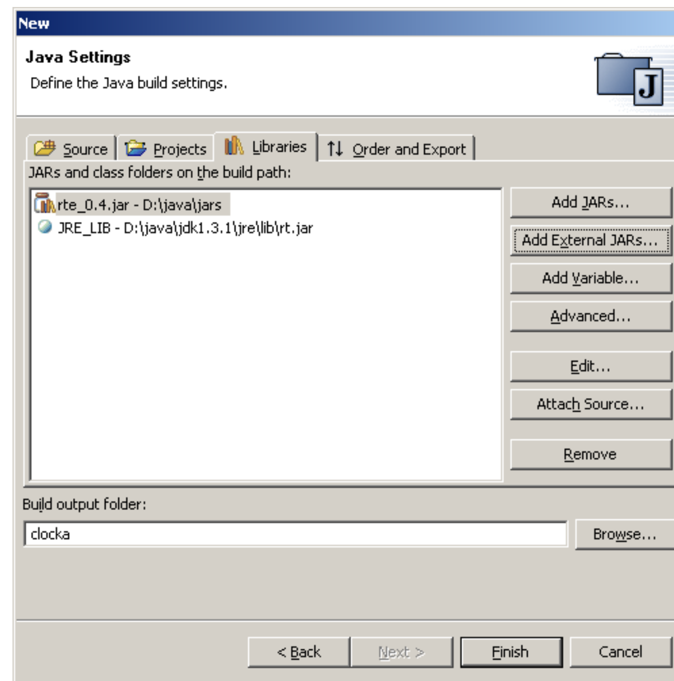

Figure 2.3: Make sure that the PECOS RTE is on the classpath

By right-clicking the project (make sure that you are in the Java perspective) you can modify the properties of the `clock` project. Most of the properties are Eclipse specific. However, the `Pecos Java Project Properties` are added by the PECOS Tools, please select these (see Figure 2.4. Now we can specify the package into which the automatically generated files are put. The default `org.pecos.generated` is fine for this example. Then we have to specify the top component of our system.

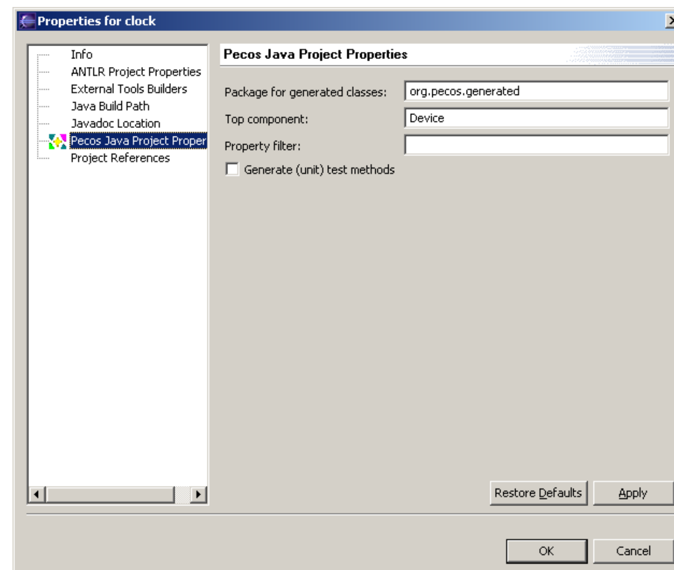

Figure 2.4: PECOS Java Project Properties

In the "top component" field we can specify the top component of our application. When nothing has been filled in, the PECOS tooling just picks one. In our example, the top component is `Device`. The property filter and generation of (unit) test methods allow more advanced uses of the code generator but we will ignore them for the moment.

As a final step before we are going to specify and build our components we should create some packages to contain the components and the main application. The easiest way to do this is to use the Eclipse Java support. Make sure that you are in the Java perspective and then click on the `Create a Java Package` button (or use the following menu item: `File-> New-> Package`. Now create a package called `components` and one called `main`.

2.4 My First CoCo

Now that everything has been set up, we can specify the three components of our Clock example. First of all, we are going to add the `Clock` component. Select the `Create new PECOS Java Component` button from the buttonbar (or use the `File->New` menu again). This brings up the wizard to add a PECOS Java component to our project. Make sure that you select the `/clock/components` folder, type in the name of the component (`Clock`) and specify that it is a passive component (see Figure 2.5).

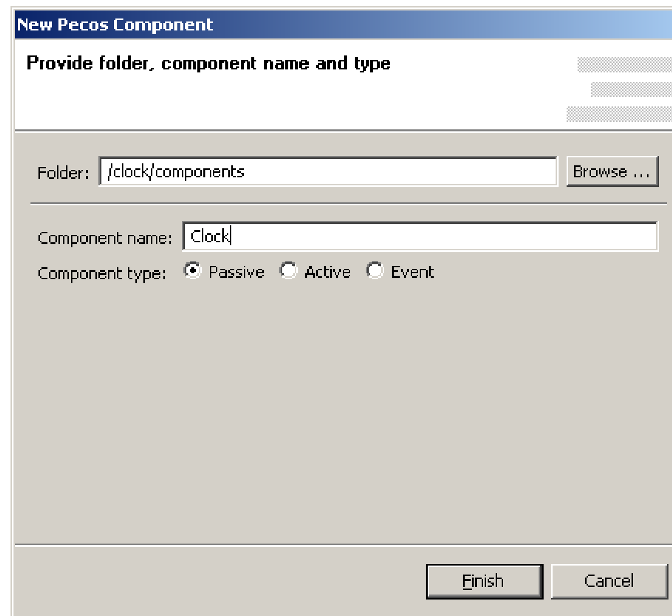

Figure 2.5: Add a new component using the PECOS Component Wizard

When `Finish` is selected, the wizard creates two files: `Clock.cm` and `Clock.java`. The first one contains the specification of the Clock in CoCo whereas the latter file is used to specify the behaviour of the component in Java. To complete the specification of the Clock component we will have to add its output port in the CoCo file. To achieve this, open the `Clock.cm` file and complete the specification until it matches the one given below:

```
component Clock {
  output long msecs;
}
```

The same series of steps should be done for the Display component. Add a PECOS Java component using the wizard, make sure that it is passive and open the `Display.cm` file. Now specify the input port of the Display component as follows:

```
component Display {
  input long time;
}
```

Finally, we have to create the CoCo specification of the Device component. We will use the same PECOS Java Wizard again, but this time the type of the component has to be set to active. After this, the `Device.cm` CoCo specification has to be updated to:

```
active component Device {
  Clock clock;
  Display display;

  connector time (clock.msecs, display.time);
}
```

The CoCo specification above describes our Clock device (see also Figure 2.6). It consists of two instances of subcomponents: `clock` and `display`. Additionally, the Display component contains a connector that links the `msecs` output port of clock to the `time` input port of display. This exactly matches the model of Figure 2.1.

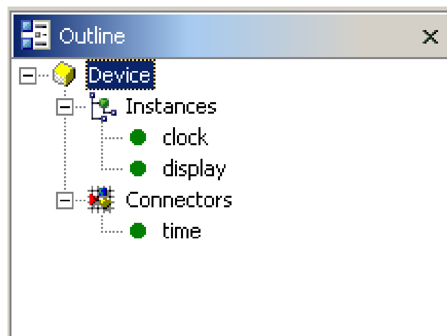

Figure 2.6: The outline view of the Device component

During the "build" of a project, which by default happens every time you save a file, the CoCo code generator generates a couple of classes. You can look into the `org.pecos.generated` package to see what has been generated after specifying these three components. For the moment, it is sufficient to only understand the very basics of what has been generated.

First of all, all components have a generated base class, e.g. the Clock component has a `ClockBase.java` class that serves as the base class for `Clock.java` in our `components` package. These base classes contain the utility getter-setter methods to access the input and output ports from Java. For example, to represent the `msecs` output port of Clock the CoCo builder generates `public void put_msecs(long val)` in `ClockBase.java`.

In addition to the generated base classes there is the `DataStore.java`. This class contains all data that is passed between the components by means of the connector mechanism. As a user you will never have to access or use the *DataStore* directly. Another generated class is the `PecosDevice` utility class. This class comes in handy, as we will see later, when we are going to initialise and run our device and set it running.

2.5 Implementing the Behaviour

Before we can actually use our components, the behaviour has to be implemented first. This should be done in the files created by the PECOS Java Wizard in the `components` package. Every passive component has two extension points that can be filled in to specify how the component

should behave at run-time: *initialize* and *execute*. For our current purposes we only need to specify the executional behaviour of the Clock and Display components. The `execute()` method in `Clock.java` should ask the system the current time in milliseconds and subsequently put this value on its output port using the msecs setter method:

```
public void execute() {
  long time = System.currentTimeMillis();
  put_msecs(time);
}
```

The behaviour of the Display component is also not too complicated. It gets the time from its input port, formats it and prints it to standard out:

```
public void execute() {
  long time = get_time();
  java.util.Date date = new java.util.Date(time);
  System.out.println("Date = "+date);
}
```

2.6 Get the Clock Going

The Device component itself has no specific behaviour. It more or less serves as an empty container that holds the Clock and Display components. However, since the Device component is an active component it does need a schedule that specifies how the Device component and its passive children run. In order to get the Clock going we therefore have to add a schedule to the project. Add a new file in the `main` package using `File-> New-> File` and call it `schedule.cm`. Edit the file until it looks like below:

```
schedule sched of Device every 1000 at 10 {
  {
    exec clock;
    exec display;
  } at 0;
}
```

The schedule specification given above (see also Figure 2.7) specifies a schedule named `sched` for the Device component (`schedule sched of Device`). The schedule is periodic and runs every 1000 millisecs (`every 1000`) at a priority of 10 (`at 10`). The schedule contains one *Job* that subsequently executes clock and display. The `at 0` means that the Job is started without any additional delay (relative start time is 0).

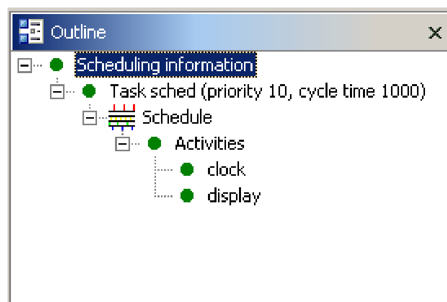

Figure 2.7: The outline view of the Device schedule

The only thing we are currently missing to get the application running is a main class. To overcome this small issue create `ClockDevice.java` in the `main` package. The implementation of this class should look as follows:

```
package main;

import org.pecos.generated.PecosDevice;

public class ClockDevice {

    public static void main(String[] args) {
        PecosDevice._initialize();
        PecosDevice.start();

        // only run for five seconds
        try {
            Thread.sleep(5000);
        } catch (Exception e) {}
        System.exit(0);
    }
}
```

The `ClockDevice` class makes use of the generated utility class `PecosDevice`. We can initialise and run the device by calling the static methods `_initialize()` and `start()` respectively.

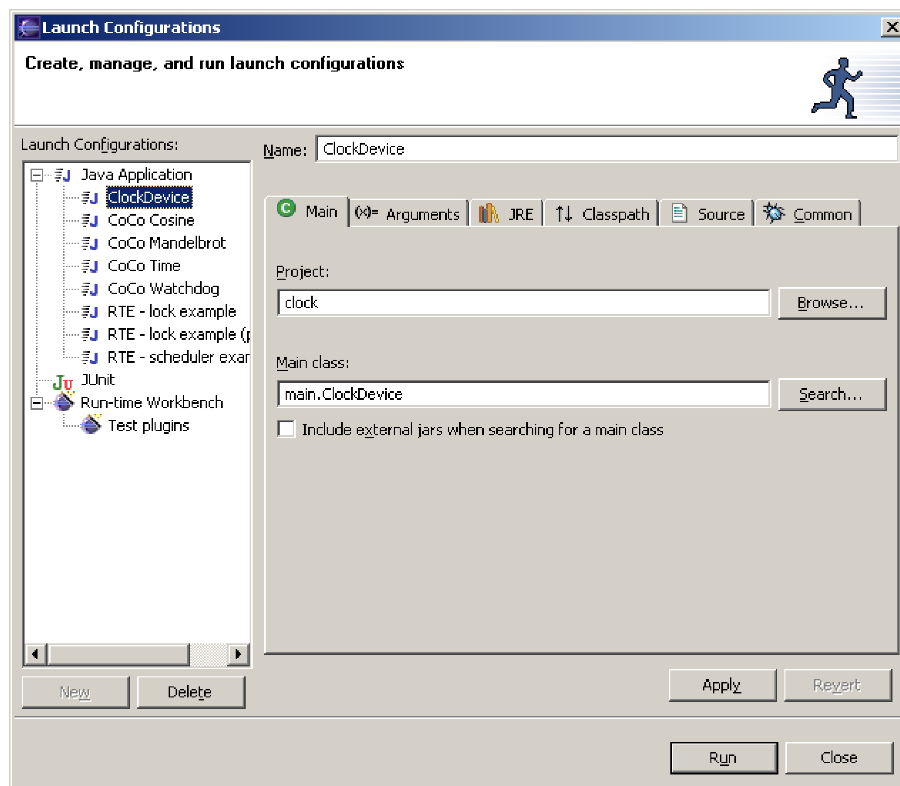

Figure 2.8: The launch configuration that starts the Clock Device

To run our newly created ClockDevice in Eclipse, you click on the small black triangle next to the running man in the toolbar and select `Run ...`. Select `Java Application` and press `New`. Make sure that the new launch configuration looks like Figure 2.8 and press `Run`. The Clock Device gets started and prints the current time at standard output as shown in Figure 2.9.

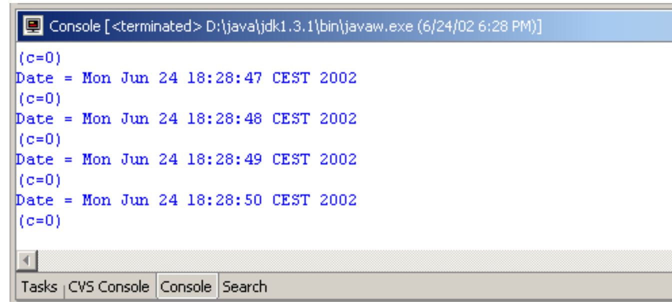

Figure 2.9: The output of the running Clock Device

2.7 Adding a Digital Display Component

After we have built the simple text-based *Clock Device* we are now going to extend it with some additional features. The goal of this example is to illustrate how an existing application can be extended by adding components.

To build a Clock Device that shows the time digitally, we have to make use of a widget toolkit. In this example we are using the *Standard Widget Toolkit* (SWT) from OTI/IBM [3]. SWT is currently available for Windows, Linux/Unix (both Motif and GTK2.0) and QNX Photon. A subset of SWT is available for Palm and PocketPC. SWT is the widget toolkit that is used in Eclipse as well. To be able to use SWT in our sample, we will have to add it to the classpath: right-click the project (in the Java perspective) and select `Properties`. Go to the `Java Build Path` property and select `Add Variable ...` Now add `swt.jar` to the classpath as shown in Figure 2.10.

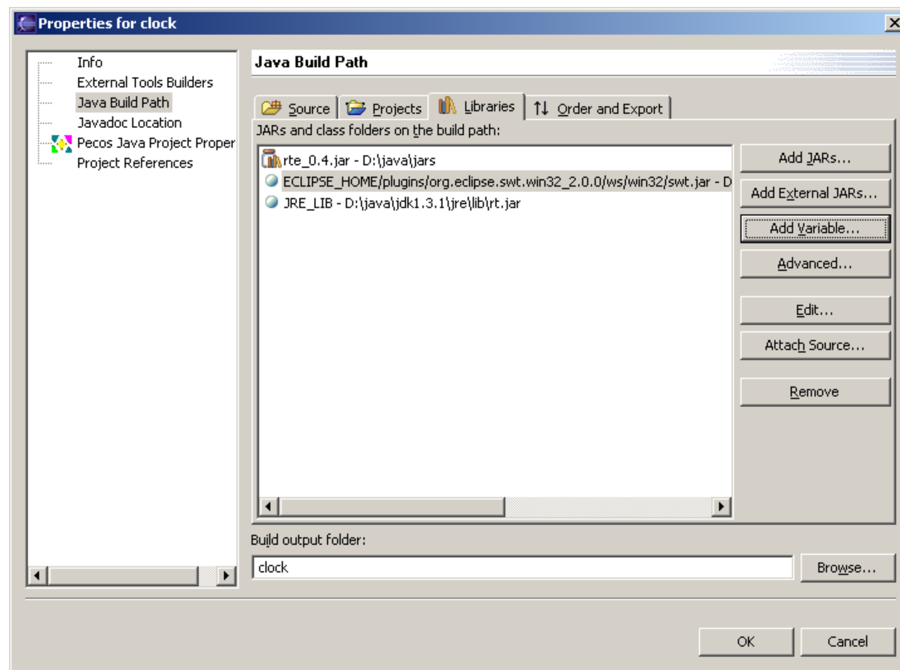

Figure 2.10: Adding the SWT toolkit to the classpath

Every graphical application using SWT needs to have an *eventloop* that handles graphical events such as mouse clicks and repaint events. The eventloop is a good candidate for reuse in multiple applications. The `EventLoop` component that we are going to use in the digital clock for example comes from another graphical sample that displays mandelbrot images.

In addition to the EventLoop component, our graphical clock needs to be extended with a component that displays the time using SWT. We will call this component `DigitalDisplay`. Figure 2.11 shows how the model of Figure 2.1 is extended with the EventLoop and DigitalDisplay components. The DigitalDisplay component has, just like the original Display component, an input port `time_in_msecs`. Additionally, it needs to know whether it can draw onto the *Canvas* provided by the graphical EventLoop component. Therefore it has a boolean input port `can_draw` that is connected to the boolean output port `started` of EventLoop that specifies that the eventloop has been started and is ready to receive graphical events. The CoCo specification of the EventLoop and DigitalDisplay components is given below:

```
active component EventLoop {
  output bool started;
}
```

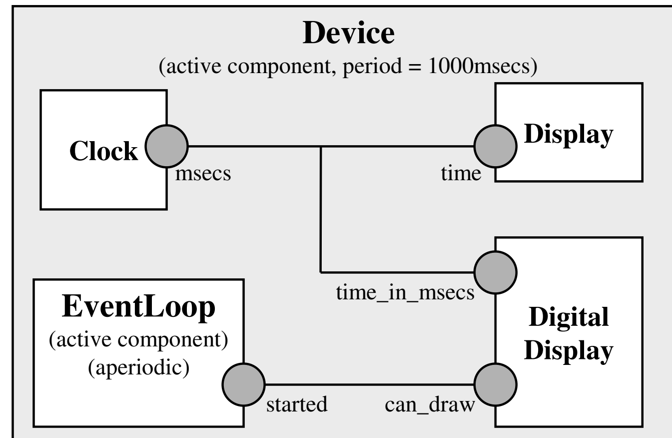

Figure 2.11: Model of the extended Clock Device example

```
component DigitalDisplay {
    input long time_in_msecs;
    input bool can_draw;
}
```

The implementation and specification of the EventLoop and DigitalDisplay components is already available on the accompanying CD. Please use `File-> Import...` to import the `EventLoop.cm`, `EventLoop.java`, `DigitalDisplay.cm` and `DigitalDisplay.java` files into the `components` package.

2.8 Update the Device to include the Digital Display Component

As Figure 2.11 shows, the updated Device contains the two new components as sub-components and has a connector to connect them. Additionally, the `time` connector should be extended to connect to the DigitalDisplay component as well. The updated `Device.cm` file should look as follows:

```
active component Device {
  Clock clock;
  Display display;
  DigitalDisplay digitalDisplay;
  EventLoop eventLoop;

  connector time (clock.msecs, display.time,
  digitalDisplay.time_in_msecs);

  connector eventLoop_started (eventLoop.started,
  digitalDisplay.can_draw);
}
```

Since we have changed the Device component, and we have added an active component to our system, we also have to update the schedule in `schedule.cm`. The fact that the EventLoop is an active component has two important implications for the schedule of our Clock Device. First of all, we will have to call *synchronize* to synchronise the connector between the `started`

and `can_draw` ports. For the moment it suffices to say that this synchronisation makes sure that connected ports between an active and another component have the same value. IN later chapters the notion of synchronisation will be described more extensively.

A second consequence of EventLoop being active is that it needs its own schedule. Since EventLoop is a *leaf* component, i.e. it does not have subcomponents, the schedule only consists of one activity that runs the component itself. The updated schedule is shown below:

```
schedule sched of Device every 1000 at 10 {
  {
    sync eventLoop;
    exec clock;
    exec display;
    exec digitalDisplay;
  } at 0;
}

schedule eventTask of Device.eventLoop at 5 {
  {
    exec;
  } at 0;
}
```

In the updated schedule, we have added a `sync eventLoop` to the main schedule to synchronise the components. Additionally there is a new schedule `eventTask` that runs aperiodically at a lower priority. This task runs the SWT eventloop.

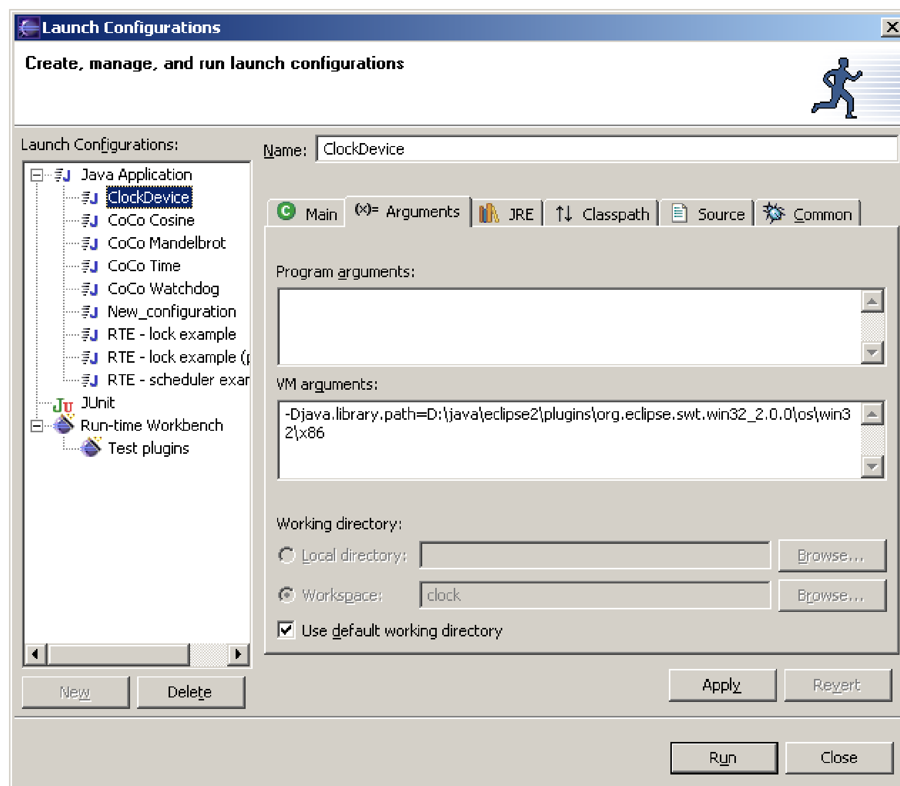

Figure 2.12: Set the VM arguments so that it can find the SWT library

When we would run the application as it is now, we would see the error that the Java VM cannot find the SWT run-time library. Therefore we have to update the "Launch Configuration" to set the Java library path in the VM arguments (see Figure 2.12). When we have done this successfully, running our Clock Device results in both textual and graphical output of the current time of the day (Figure 2.13);

Figure 2.13: Graphical output of the Clock Device

2.9 Wrap up

This chapter introduced the basic knowledge about PECOS and the available tools and technologies. By means of the *Clock Device* example we have seen the basic aspects of how to create a simple PECOS application (see also Table 2.1. In particular this chapter demonstrated the use of the Eclipse tooling and the CoCo language. The following chapters will discuss these issues and more in a lot more detail.

Table 2.1: Steps to build a simple CoCo Java application

1. Create a PECOS Java Project.
2. Build your component specification in CoCo (`component.cm`).
3. Implement the behaviour in Java (`component.java`).
4. Add a CoCo schedule for every active component.
5. Build the project (creates the base and utility classes).
6. Write a main class that uses the `PecosDevice` utility class.
7. Create a launch configuration and run the application.

Chapter 3

Component Model

3.1 Introduction

In this chapter, we present the PECOS Component Model. It is a specific component model that addresses the constraints of field devices and is the foundations for ABB Component Development. To make the user familiar with the introduced terminology, we explain the concepts using the example presented in the Chapter 2.

3.2 Structural Overview

As we saw in the example described in the Chapter 2, we can identify three main entities in the PECOS Component Model: *components* , *ports* and *connectors*. We will see the main features of these entities and how we can map them based on the example of the `Clock Device`. In the next section we then discuss the runtime semantics of the model.

3.2.1 Components

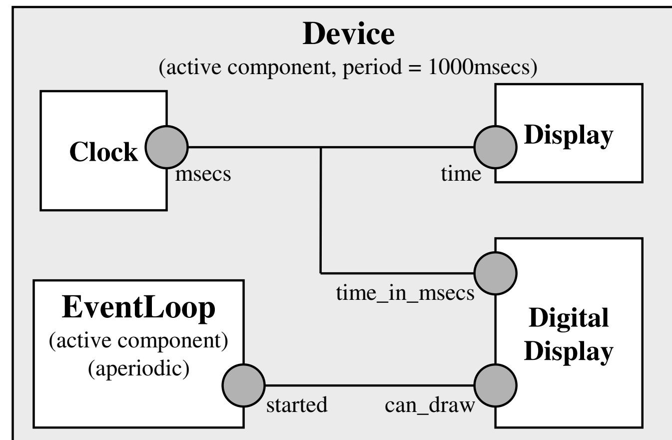

Figure 3.1: Model of the extended Clock Device example

A component is the core entity in this model. Components are used to organise the computation and data into parts that have well-defined semantics and behaviour. Figure 3.1 shows the component model again for the *Digitial Clock* example from section 2.7. We see five components:

the `Clock`, the `Display`, the `DigitalDisplay`, the `EventLoop` and the `Device`. This last one is a kind of container for the rest of the components.

Every component has a *name*, a number of *property bundles* (used to store meta information of components, such as worst-case execution times or scheduling information), a set of *ports*, and a *behaviour*. The *behavior* 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. Depending on how the action is triggered and where it is run, we distinguish different kinds of components.

Passive Component A passive component does not have its own thread of control. Passive components are typically used to encapsulate a piece of behaviour that executes synchronously and completes in a short time-cycle. In our example, the components `Clock`, `Display` and `DigitalDisplay` are all passive components.

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). In our example, the component `Device` and `EventLoop` are active components.

Event Component An event component is like an active component, but the execution of the behaviour is triggered by an event. Certain pieces of hardware frequently emit events, such as motors that give their rotation speed. Whenever the event fires, the behaviour is executed immediately. This is for example used to make timers that have to fire at a certain moment.

The *Digital Clock* example uses active components (`Device` and `EventLoop`) and passive components (all the other ones). There is no event component in that example.

3.2.2 Ports

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). Looking more closely, a port is a reference to data than can be read and written by a component and enables a component to be connected to another component (through a connector). A port is specified with 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.

In the *Digital Clock* example, we identify the following ports:

- `msecs` for the component `Clock`: This port contains the current time in miliseconds. It is an `out` port (because it provides the time of the component), and has as type `long`.
- `time` for the component `Display`: This port formats the time and prints it to standard output. Its direction is `in`, because it receives the time for the component, and has as type `long`.
- `started` for the component `EventLoop`: This port specifies that the eventloop has been started. It has as type `bool` and it is an `out` port that sends a signal to activate the `DigitalDisplay`.

- `can_draw` for the component `DigitalDisplay`: This port is used to indicate that the display is ready to receive graphical events. It is an `in` port of type `bool` that receives a signal to indicate that the display is ready to receive graphical events.
- `time_in_msecs` for the component `DigitalDisplay`: This port receives the time to display it. Its direction is `in`, and its type is `long`.

3.2.3 Connectors

It is clear that the ports act as providers and receivers of data. To fulfill this characteristic, we need to connect them. A `connector` describes a data-sharing relationship between ports. It is described with the following features:

- a name,
- a type (that has to be compatible with the port types), and
- a list of ports it connects.

In the example, the ports of the components are connected using the connector `time` that connects the ports `clock.msecs` and `display.time` of the components `Clock` and `Display` respectively. In the extended example to include the `DigitalDisplay` component, the connector is extended to connect also the `digitalDisplay.time_in_secs`. There is also another connection made using the connector `eventLoop_started` that connects the ports `eventLoop.started` and `digitalDisplay.can_draw`.

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

3.2.4 Composite Components

Every component can contain (connected) subcomponents. When it contains subcomponents, we call it a *composite component*. When it has no children, we call it a *leaf component*. The subcomponents of a composite component are not visible outside the composite component. A composite component can have external ports that are connected to selected ports of its subcomponents. In the *Digital Clock* example, the composite component is the component called `Device` and the subcomponents are the components `Clock`, `Display`, `DigitalDisplay` and `EventLoop`. Note that the composite component `Device` has no external ports because it represents the application to be run.

A composite component is responsible to provide a *schedule* that specifies the order in which its own behaviour and the behaviour of the subcomponents is run. In the example only the `Device` component has to do this, since it is the only composite component (see Sections 2.8 for the description of the schedule).

3.2.5 Properties and Properties Bundles

Meta information of a component, such as memory consumption or worst-case execution time, is expressed using *properties* and *property bundles*. A *property* is a tagged value, where the tag is used as an identifier. For example, we could specify the `cycletime` of a component with the tag `cycletime` and the value `100`.

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. For example, scheduling information is expressed using *cycle time* and *worst-case execution time*.

Properties bundles are used by different PECOS tools, such as the Composition Rule Checker, the Schedule Generation tool and the Schedule Verification Tool.

3.2.6 Parent

The component structure implied by the model is always hierarchical. The top is an active composite component (the device) that contains a number of subcomponents . Every one of these subcomponents can again contain components, with or without subcomponents. Because of this hierarchical structure we introduce a scope for components.

The *parent* of a component X is the immediate composite component within which X is nested. In the example the parent for the components `Clock`, `Display`, `DigitalDisplay` and `EventLoop` is the `Device` component.

3.3 Execution Model

The previous section explained the structural part of the model (components, ports and connectors), but does not talk about its execution semantics. The goal of this section is to make clear what happens at runtime. We first explain how data gets synchronised between components running in different threads and then describe the actual runtime semantics.

3.3.1 Data spaces

Before we look at the complete runtime semantics we need to discuss how data gets synchronised between components. Remember that a field device has always a hierarchical structure. The top is the field device itself, that is an active composite component containing a number of components. Every one of these components can contain subcomponents, and so forth. The result is a tree of components.

In this tree of components, some components are passive, while other ones are active or event components. Like we have seen in Section 3.2.1, active and event components have behaviour that runs in their own thread of control. When two active components provide ports that are connected by a connector, they can both read and write to the data residing in the ports all the time. Hence the model needs to assure that data cannot get corrupted due to two simultaneous write operations from components in different threads.

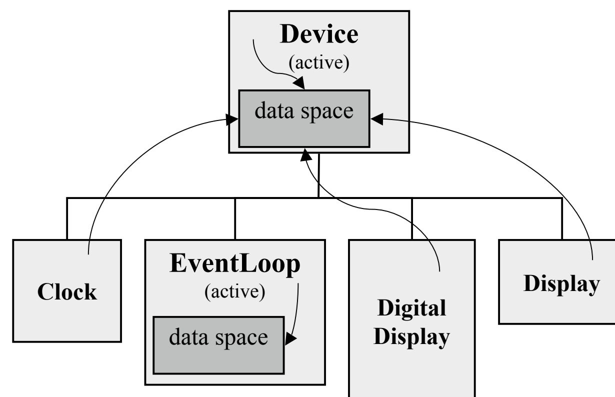

Figure 3.2: Tree view of the component model of the *Digital Clock* example. It shows the components and the private data spaces of the two active components in the example (*Device* and *EventLoop*).

To solve this problem every active and event component is equipped with its own private *data space* in which it can work, separate from the rest of the world. At specific intervals, this private

data space is then synchronised with the one of the parent. During this synchronisation the contents of the private data space are brought in sync with the data space of the parent, so that at the end of the synchronisation phase both data spaces contain the latest data. The private data space is used by the execution method of an active component, and by all of its subcomponents (unless they are active, in which case they work and synchronise their own data space again). Figure 3.2 shows the tree for the *Digital Clock* example of Section 2.7. It shows the `Device` as the root of a tree with four leaves (one for each subcomponent): `Clock`, `Display`, `EventLoop` and `DigitalDisplay`. All subcomponents but `Eventloop` are passive, and they use the data space provided by `Device`. The `EventLoop` component uses its own data space. Any passive subcomponents of `EventLoop` would use the data space of `EventLoop`. Any passive subcomponents of `Display` would use the data space provided by `Device`.

3.3.2 Execution and Synchronisation Behaviour

We just said that active and event components need to be able to synchronise their data space with the data space of their parents. So from an execution point of view there are two different behaviours associated with active and event components: *execution behaviour* and *synchronisation behaviour*. *Execution behaviour* determines the action that is performed when the component is executed. *Synchronisation behaviour* is responsible for synchronising the data space of the active or event component with that of the parent. Note that the root component is an exception to this since it only has subcomponents and no execution nor synchronisation behaviour.

3.3.3 Runtime Semantics

Now that we know how the data synchronisation is handled, we see that the runtime semantics of the model are governed by simple rules:

- The behaviour of a passive component is executed in the thread of its parent component.
- Synchronisation behaviour for active and event components is executed in the thread of the parent component.
- Active and event components execute their subcomponents and their own execution behaviour in their own thread of control.
- Every composite component has to provide a schedule for its children.

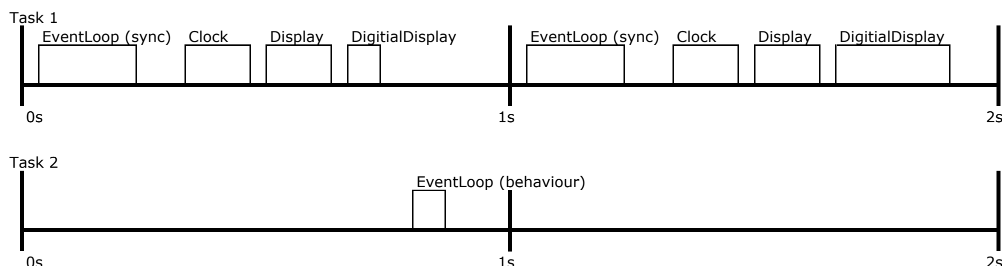

Figure 3.3: Possible Execution trace for the *Digital Clock* example.

Figure 3.3 shows an example of 2 seconds in a possible execution of the component model from Section 2.7. Because there are two active components in the Digitial Clock example, we see two tasks: one associated with the Device component (task 1), and one associated with the EventLoop component (task 2). Since the period of the device is 1000 ms, this means two periods in this device. In the first period of task 1, the synchronisation behaviour of the EventLoop is executed, followed by the behaviours for the passive components. When they are finished

executing, the lower priority task2 executes the behaviour of the EventLoop component, that initializes the eventloop to handle graphical events. In the second period of the device more or less the same happens for task 1: the synchronisation and behaviours are executed. Nothing happens in task 2 in this run.

3.4 Summary

In this chapter, we explained the main concepts related to PECOS Component Model. Firstly, we have shown the different entities comprised in the model and how they are identified in the example presented in the Chapter 2. We also explained the foundations of the execution model of the model and how the different components -according to their features- are synchronised. In the next chapter we see how to use the component description language CoCo to write down component specifications.

1. The structural elements of a model are *Components*, Ports and Connectors.
2. There are 3 kinds of components: *Passive*, *Active* and *Event*.
3. Every component (regardless of its type) is either a *composite* (when it has subcomponents) or a *leaf* (when it has no subcomponents).
4. Ports indicate data sharing points. They are typed, and have as direction *in*, *out* or *inout*.
5. Connectors connects ports to express data sharing relationships.
6. Active and event components have two kinds of behaviour: *execution behaviour* and *synchronization behaviour*.
7. The execution semantics are given by the following rules:
    - The behaviour of a passive component is executed in the thread of its parent component.
    - Synchronisation behaviour for active and event components is executed in the thread of the parent component.
    - active and event components execute their subcomponents and their execution behaviour in their own thread of control.
    - Every composite component has to provide a schedule for its children.

Chapter 4

The CoCo Language

By Michael Winter

4.1 Introduction

CoCo stands for **Co**mponent **Co**mposition Language. It is an implementation of the PECOS component model which has been introduced in the previous chapter. This chapter gives a brief overview of the CoCo language constructs and shows how CoCo is used to specify PECOS components and whole PECOS field devices. Therefore, the Clock example from chapter 2 is revisited and new language constructs, not present in the example, are introduced.

4.2 Keywords

The CoCo language has got the following reserved keywords:

active, at, component, connector, exec, extends, event, every, has, input, inout, of, output, passive, port, properties, schedule, sync, type

The following words are additionally reserved words for CoCo base types:

byte, bool, char, short, int, long, float, double, void

Keywords and reserved words cannot be used for identifiers. The purpose of the different keywords will be explained in the sequel of this chapter.

4.3 Components

The most important purpose CoCo is used for is the specification of PECOS components. A very simple example of a PECOS component is the `Display` component taken from chapter 2, which is specified in CoCo as

```
component Display {
  input long time_in_msecs;
}
```

A component specification in CoCo always starts with the keyword **component** followed by the component's name. This name has to be unique within a PECOS project. The body of the component specification follows the component's name and is enclosed in braces. In the above example, this component body contains only the declaration of an input port called `time_in_msecs`;

More generally, a component's declaration can look like the following:

```
[passive | active | event] component componentName
[has propertySetNames, ...] {
  // Properties Section (optional)
  properties{...};
  // Ports
  [in | out | inout] dataType portName;
  // Component Instances
  componentName instanceName;
  // Connectors
  connector(instanceName.portName, ...)
}
```

As we have already seen in 3.2.1, the PECOS component model defines three different types of components: *active*, *passive* and *event* components. Active and Event components are distinguished from passive components, like e.g.the `Display` component above, by putting the **`active`** or **`event`** keywords in front of the component specification, respectively.

Next, a component can specify a list of *property sets* following the **`has`** keyword. Property sets will be examined in detail in sections 4.6.2. Note, that property sets are sometimes also called *property bundles*. In addition to property bundles, one may specify properties in the *properties section* of a component. This section starts with the **`properties`** keyword and is the first thing that may appear in a component's body. Properties will be looked at in section 4.6.

Besides that, a component's body may contain *ports* declarations, explained in section 4.4, *component instances*, explained in section 4.5.1, as well as *connectors* which will be looked at in section 4.5.2.

4.4 Ports

Let us begin with the ports. As already known from chapter 3, the PECOS component model is data–flow oriented: components exchange data with their environment through ports. More precisely, a component's ports are the only means of a component to communicate with it's environment (that means with other components). Thus a component's interface is given by the set of ports it provides.

Data ports can be declared in a component's body. For instance, the `Display` component declares one single port:

```
input long time_in_msecs;
```

This port has name `time_in_msecs` — which has to be unique within the components body. The keyword **`input`** indicates the dataflow direction of the port, which in this case is a data flow from the component's environment *into* the component.

Besides its name and data flow direction, every port has got a data type assigned, like **`long`** in this case. The specified data type restricts the data which may flow through the port to be of that type. For details about CoCo's data types please have a look at section 4.8.

Besides input ports, there is also *output* ports (keyword **`output`**, through which data can flow out of a component. We have already encountered an output port in the introductory example in the `Clock` component:

```
component Clock {
  output long msecs;
}
```

In addition to input and output ports, there are also *inout* ports (keyword **`inout`**), which allow data–flow in both directions, into and out of a the component through the same port.

4.5 Composite Components

The two components (Clock and Display) we have seen so far are so–called *leaf* components. They do not contain other components, but are directly implemented in Java or C++. How this is done has been presented in the introductory example. In contrast, composite components are not implemented that way but are build form other components. In particular, every PECOS field device application is modelled as a composite component.

4.5.1 Component Instances

Composite components contain instances of other components. For instance, the Device component shown below is a composite component containing the two component instances clock and display of type Clock and Display, respectively.

```
active component Device {
  Clock clock;
  Display display;

  connector time (clock.msecs, display.time_in_msecs);
}
```

4.5.2 Connecting Component Instances by Connectors

In order to let these components exchange data, their ports (declared in the respective component specifications) have to be connected. In our example, the msecs port of component(–instance) clock and the time_in_msecs port of component(–instance) display are connected by a connector named time.

Note, that msecs is an out–port while time_in_msecs is an in–port. This information is not visible within the connector declaration:

```
connector time (clock.msecs, display.time_in_msecs);
```

Note also that the order of specification is of no importance. This stems from the fact, that CoCo allows to connect more than two ports by one connector. We will see this in more detail in section 4.8, but for the moment we stay with only two ports for simplicity.

A connector may only connect two ports, if the date type of the in–port is compatible with the data type of the out–port. In our example, both ports are of type **long** and are therefore compatible, of course. More on data types in CoCo is presented in section 4.8 below.

More importantly, ports of components can only connected, if they reside within the same enclosing component. Connectors may not cross component boundaries!

Connectors in CoCo have got a name which is unique within the enclosing component. This name is not referenced from within CoCo, but merely simplifies arguing about CoCo specifications for the developer.

4.5.3 'Upward' Connectors

Besides connecting component instances within a composite component, connectors server another purpose in CoCo. For instance, imagine you want to reuse Device itself as a sub–component within some other component — let us call it Bigger_Device — and that in Bigger_Device you need access to the msecs port of component clock in Device.

But this is not possible! CoCo only allows access to the ports of direct sub–components in Device. For instance, no access to msecs from Bigger_Device is possible. Thus the intended reuse is not possible with the actual specification of component Device!

A solution to this problem would be, to define a new Device component — let us call it New_Device — and to introduce an additional port (let us also call it exported_msecs) into

this component which is connected to the msecs port of component clock. Thus, New_Device would look like the following:

```
active component New_Device {
  // new port msecs
  output long exported_msecs;

  Clock clock;
  Display display;

  connector time (clock.msecs, display.time_in_msecs);

  // new connector export_msecs
  connector export_msecs (clock.msecs, exported_msecs);

}
```

Now, by using New_Device instead of Device, access to the port of sub–component Clock is possible by accessing the exported_msecs port of component New_Device.

4.5.4 Semantics of Connectors

Let us have a closer look at what it means to connect two ports by a connector.

Shared Variable Semantics

The (informal) semantics of CoCo connectors is the one of a data variable. A port being part of a connector declaration denotes access to this variable.

Access to the shared variable is purely synchronous. That means, there is no risk of conflicts due to concurrent access to the variable by several tasks.

A CoCo connector holds *exactly one data item at any point in time* — it does not have a buffer semantics! Thus, writing to or reading from a port has got the following effects:

- If a component writes a value onto an out–port, it actually writes this value to the associated shared variable.
- If a component reads from an in–port, it actually reads the value stored in the shard variable.
- If a component reads from an in–port multiple times, it will read the same value — assuming that no other component has changed the value in the meantime.
- If a component writes to an out–port several time, it will override the value stored there each time.

Connecting Multiple Ports

Connectors are not limited to connect only two ports, but may connect an arbitrary number of ports that lie in the same scope. This results in more than two ports being associated with the same shared variable. Access to any of the connected ports will access this shared variable.

In this case, the data type of each in–port has to be compatible to the data types of all out–ports in the connector. Otherwise type safety could not be guaranteed.

Connectors Sharing Ports

Different connectors that share a common port represent the same shared variable. For instance, the connector

```
connector time (clock.msecs, display.time, digitalDisplay.time_in_msecs);
```

from section 2.8 could also have been written in the following way:

```
connector time (clock.msecs, display.time);
connector time_2 (clock.msecs, digitalDisplay.time_in_msecs);
```

as they have got the port `clock.msecs` in common.

Note that this rule is not only true for connectors residing inside the same composite component but is valid globally for the entire device — as long as no border of an active or event component has to be passed (these are synchronisation borders — see following section for this exception!).

Synchronisation Boundaries

There is one more important thing one has to know about connectors. This issue has to do with the fact that active and event components run in their own thread of control and thus in parallel. As a consequence, synchronisation has to be taken into account at the borders of active and event components.

In PECOS, this is done by decoupling the connectors connecting to an active (or event) component from the inside from those connecting from the outside. This is thus an exception to the last paragraph: connectors connecting form the inside to the port of an active component denote a different shared variable than connectors connecting from the outside to this port! Data only can cross this border within the synchronize() handler of the active (or event) component!

For the programmer this has got the effect, that data is not propagated immediately over synchronisation borders, but is only available after the *synchronisation* action of the involved active or event component has been performed.

4.6 Properties

Properties serve to specify functional and non–functional features of components, like e.g. default settings for port values or the memory consumption of a component. Properties provide Meta–Information to development tools (Timing Verification, Schedule Generation, Testing Tools) which can inspect the component model of a device for different purposes during the development process.

Properties can be attached to almost all elements of the PECOS component model: components, component instances, ports, connectors, data types and schedules. The following two sections present the CoCo syntax used for that.

4.6.1 Simple Properties

The following code–snippet shows how properties can be attached to components (using our `Clock` example):

```
// properties for a component
component Clock {
  properties {
    memSize = 32;
    description = "This is my first clock component.";
  }
}
```

The keyword **properties** marks the section in the body of the component declaration, where properties can be declared. Note: If a component defines a properties section, then this has to be the *first* thing to be declared within the component body. In the above example, component `Clock` defines two properties attached, namely `memSize` and `description`. At the same time, values are specified for these two properties.

These values are used as default values for all component instances of this component type. But these default values can be overridden on a per–instance basis, as shown in the example below. In component `Device` the `description` property is set to a different value.

```
// changing default values & declaring properties on component instances
active component Device {
  Clock clock {
    description = "This is the first use of my first clock!";
    some_specific_value = 3;
  }
  Display display;
  connector time (clock.msecs, display.time_in_msecs);
}
```

This example also demonstrates, how properties can be directly attached to component instances. `some_specific_value` in component instance `Device` is only attached to this specific instance, while other component instances of type `Device` will not have this property.

Properties cannot only be attached to components, but also to ports and connectors. The following example demonstrates how the two properties `minValue` and `maxValue` are attached to port `exported_msecs` of component `Device`. Also an example of setting properties on a connector are given.

```
component New_Device {
  // properties on a port
  output long exported_msecs {
    minValue = 0;
    maxValue = 32000;
  }

  Clock clock;
  Display display;

  // properties on a connector
  connector exported_msecs (clock.msecs, exported_msecs) {
    prop_1 = 0;
    prop_2 = 8;
}
```

In order to being able to change the default values of property settings of ports and connectors on a per–instance level as this is possible for component instances, the following syntax is used:

```
component Some_other_Device {
  // change port and connector properties on a per-instance level
  New_Device newDevice {
    newDevice.exported_msecs.minValue = 2;
    newDevice.exported_msecs.maxValue = 15;
    newDevice.exported_msecs.prop_1 = 3;
  }
}
```

This changes the port and connector properties of component instance `newDevice`. Alternatively, these settings can also be performed in the **properties** section.

4.6.2 Property Bundles

Properties can be structured in so–called property bundles. These bundles group properties that semantically belong together. An example of such properties are the minimal and maximal values of a port. These properties can be bundled together into a property bundle `PortValueProperties` as follows

```
properties PortValueProperties {
  minValue;
  maxValue;
}
```

With this definition, the specification of port `exported_msecs` above can be written more elegantly in the following way:

```
output long exported_msecs has PortValueProperties {
  PortValueProperties.minValue = 0;
  PortValueProperties.maxValue = 32000;
}
```

Property bundles can be attached to components, ports and connectors using the **has** keyword. Setting the values of properties being grouped in a property bundle is done by stating the property bundle name followed by the property settings in braces as shown in the following example with the `exported_msecs` example.

```
component New_Device  {
  output long exported_msecs has PortValueProperties {
    PortValueProperties {
      minValue = 0;
      maxValue = 17;
    }
  }
```

In the same way, property bundles can be attached to components, connectors, etc. using the **has** keyword.

Note: When declaring properties in components and property bundles, one may or may not provide a default value for a property. Properties, without a default value (like in the `PortValueProperties` example), which are not set later on a per–instance basis are skipped by CoCo and are not visible to any tools.

4.7 Schedule Specification

As we know from chapter 3, *every active component* in a PECOS device runs in it's own task and has to specify a schedule for it's subcomponents. An example for a schedule has already been shown in the introductory example in section 2.6. Here it is again:

```
schedule sched of Device every 1000 at 10 {
    {
        exec clock;
        exec display;
    } at 0;
}
```

Here, `sched` is the name of the schedule and `Device` is the name of the (only) active component to which the schedule belongs. The **every** keyword is followed by the cycletime of the schedule in milliseconds, which means that the whole schedule is executed every 1000 milliseconds. Schedules which do not execute cyclically but only once, omit the **every** keyword. Finally, the **at** keyword is followed by the priority at which the schedule is executed.

Within the body of a schedule, one can now define a number of jobs and when these jobs are executed. A job consists of a list of activities. An activity stands for the execution of an action on a component. There are two possible actions: either the execution of the `exec()` or the `sync()` handler of a component. Figure 4.1 shows the dependency between tasks, jobs, activities as a class diagram.

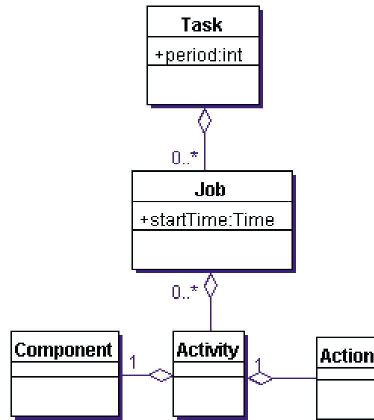

Figure 4.1: Class Diagram of Tasks, Jobs and Activities

For instance, the above example contains a single job (enclosed in braces), which consists of the two activities **exec** `clock` and **exec** `display` — say, the consecutive execution of the exec handler of component `clock` and the exec handler of the component `display`.

Finally, the **at** keyword behind the job states that this job is executed at the relative starting time 0 within the cycletime of the schedule.

Normally, a schedule will consist of more than one job. E.g., if one wants to execute the exec handler of component `clock` at relative staring time 0, while the exec handler of component `display` is invoked at relative starting time 200, one would need to specify two jobs in the following way:

```
schedule sched of Device every 1000 at 10 {
    {
        exec clock;
    } at 0;

    {
        exec display;
    } at 200;
}
```

Also it may be the case, that identical jobs are executed at different points in time (that means at different relative starting times within the same schedule). In order to avoid writing the same code over and over again, one can explicitly name jobs.

For instance, assume one wants to execute the job from our first example two times within one schedule, say at relative starting times 0 and 500. This can be done in the following way:

```
schedule sched of Device every 1000 at 10 {
    // job definition
    job_1 = {
        exec clock;
        exec display;
    };

    //job execution
    job_1 at 0;
    job_1 at 500;
}
```

The solution was thus, to give the job the name `job_1` and to use this name to execute the job at the two relative starting times 0 and 500.

Note, that CoCo does not allow to mix job declarations and execution. But all job declarations have to stand before the execution list.

Moreover, schedules can also have properties. Like with components, one has to use the **properties** keyword inside the body in order to specify properties. Property bundles are attached with the well known **has** keyword.

4.8 Data Types

Data types define data structures which can be transferred between components. Data types in CoCo are distinguished into base data types (also called built–in types) and user data types. The following built–in types are supported:

byte, bool, char, short, int, long, float, double, void

The meaning of these base–types is not specified in CoCo, but is deferred to the implementation language used, which can be either Java or C++ at the moment.

CoCo supports arrays built from these base types (as well as user–types presented below). For example, the following declaration specifies an array of length 50 with element type char:

```
char[50]
```

Base types and arrays can be used to build more complex user types in the manner of C++ structs. The following example shows a simple user data type for a person:

```
type Person {
  char[50] forename;
  char[50] surname;
  short age;
}
```

User types can be extended with fields by use of the **extends** mechanism. For instance, the following user type definition extends the Person data type with the new field officenumber:

```
type Worker extends Person {
  int officenumber;
}
```

The semantics of the extends construct is a simple inclusion of the fields declared in the data type being extended to the extending data type. Overriding of earlier defined fields with the same name is not possible.

4.9 Summary

In this chapter we have presented the different language constructs of the CoCo language. It has been shown, how CoCo is used to specify PECOS components and how to build composite components from component instances and connectors. Then, we have introduced component properties and data types. A brief summary of the language features presented here is given in Figure 4.9.

After having read this chapter, you should be able to understand arbitrary CoCo specifications and be prepared to write your own ones. Nevertheless, this chapter could not give an exhaustive presentation of the language syntax but concentrated on the most important aspects. For any additional information, please refer to the complete CoCo Grammar in Appendix A and to the CoCo Deliverable [2].

1. Components

```
[PASSIVE | ACTIVE | EVENT] COMPONENT componentName
[HAS propertySetNames, ...] {
  // Properties Section (optional)
  PROPERTIES{...};
  // Ports
  [IN | OUT | INOUT] dataType portName;
  // Component Instances
  componentName instanceName;
  // Connectors
  CONNECTOR(instanceName.portName, ...)
}
```

2. Schedule Specifications

```
SCHEDULE scheduleName OF activeComponentName
[EVERY cycleTime AT relativeStartingTime] {
  // Job Definitions
  jobName = {
    EXEC activity;
    ...
  }
  // Job Executions
  jobName AT relativeStartingTime;
  ...
}
```

3. Property Bundles

```
PROPERTIES propertySetName {
  propertyName [= defaultValue];
  ...
}
```

4. User Data Types

```
TYPE typeName [EXTENDS typeName] {
  baseType identifier;
  baseType[integer];
  ...
}
```

Figure 4.2: Summary of CoCo's top–level syntax elements.

Chapter 5

Component Behaviour

By Reinier van den Born

5.1 Introduction

In the previous chapters we have learnt about the PECOS model and how components can be specified in CoCo. But that specification is actually only half the story. It only allows the description of a component's interface and its composition. It does not specify what the component is to do, its behaviour.

This behaviour can have two origins: according to the model a component can have its own "local" behaviour and a composite component can have behaviour through its subcomponents. The one doesn't exclude the other. However, as there are no compositional rules for behaviour this combined behaviour is more or less undefined. Therefore it is actually easier to see an application as an unstructured collection of bits of local behaviour that are orchestrated separately to perform the desired function. The composition found in composite components then just serves to specify communication channels (ports and connectors) and set synchronisation boundaries (at active and event components).

In this chapter we will look at how local behaviour is specified and how it can be coordinated into performing more complicated tasks. This will not be an exhaustive treatment but it should give you a good feeling for what makes PECOS applications tick.

Language Before we start we should mention that the PECOS system actually allows a choice of programming languages to use for your application. This language will be referred to as the *implementation* or in the context of code generation the *target* language. Currently both Java and C++ are supported. Although the languages have their obvious differences there is nothing special about either of them in the PECOS context. Applications are built the same way in C++ as in Java. The CoCo is the same; the structure of the application is the same; it is only the specification of behaviour in the implementation language that is different. In this chapter the examples will be in C++.

During our discussions we will often refer to the *system*. By this we mean that part of the application that is not written directly by you. So this does not just include something like the operating system, but also any code that is generated. Even code generated from CoCo specifications that are provided by you. What is meant is basically the environment in which you do your programming.

Structure This chapter is organised as follows. Section 5.2 first gives a quick look at the general structure of a PECOS application. Then Section 5.3 explains how components are implemented in the target language and how behaviour fits in. This is followed by 5.4 where the different kinds

of behaviour are examined in depth. Section 5.5 looks at the problem of getting all these bits of functionality to run together correctly. Finally 5.6 describes the last bits that need to be done to get an application to run. And finally in Section 5.7 the story is wrapped up.

5.2 Application Structure

We have seen in the introductory example in Chapter 2 that an application can be specified by a combination of CoCo descriptions and target language classes. The CoCo parts were translated into classes that, together with the ones provided directly, were compiled into a runnable application. What may have been less obvious from the example is that to run them these classes are linked with a library that is called the *Run-Time Environment* (RTE).

This RTE is a standard library that comes with PECOS. It has three purposes:

- to provide an abstraction level over the Real-Time Operation System (RTOS).
- to provide some base classes for the generated code
- to provide some standard functionality to run and internally synchronise applications.

Note that while the RTE interface is standard its implementation isn't. So for a given hard- and software platform, you will need a specialised version. On the other hand the RTE is application independent: once you have one for a given platform, you can build any PECOS application on top of it.

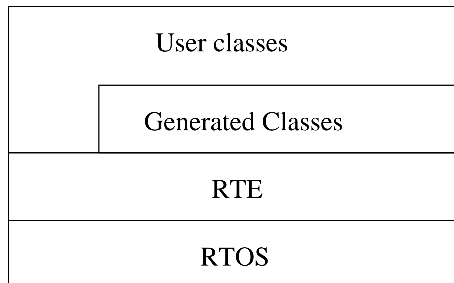

Figure 5.1: Application Structure

One advantage of this RTE is of course is that it allows components to be kept platform independent. Meaning that they can be shared more easily between projects. Of course this may not always be possible for components that interface with some specific hardware.

Coming back to our application we find that, together with the operating system it consists of four layers: the RTOS, the RTE, the generated and the user provided code. This is depicted in Figure 5.1.

Other details of the RTE are not really important here. For the interested an extensive description can be found in [5]. Likewise [7] provides a comprehensive specification of the code generation.

5.3 Component Structure

Components are built from a three level class hierarchy. Figure 5.2 below shows the relation between the classes and their most important members. The classes are the following:

- the top level is the *component type base class* which is provided by the RTE. As the name suggests there is one such class for each component type (passive, active, and event).
- the intermediate level, the *component base class*, is generated from the CoCo specification of the component.
- at the bottom level there is the *component class*. This is the one that is provided by the designer (you). For instance by completing the class generated by the new component wizard.

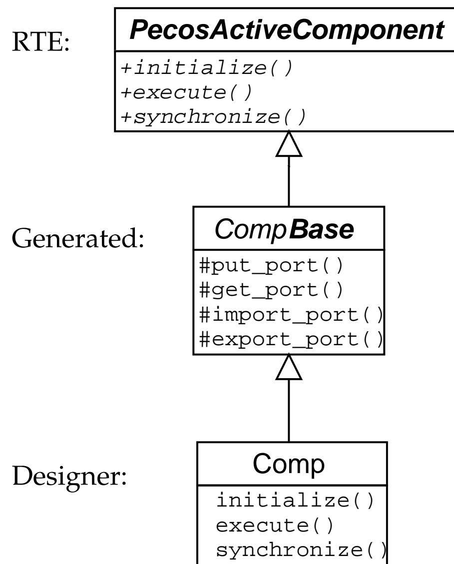

Figure 5.2: Classes for active component `Comp`

5.3.1 Component Type Base Classes

The component type base classes are provided by the Run-Time Environment (RTE) class library. They define the interface needed to run components, that is, to execute their behaviour. This interface consists, depending on the type of the component, of two or three abstract methods. Together these methods, once implemented, constitute the local behaviour of a component. As we have seen, the actual implementation of these methods is to be given in the component class, and is therefore specified by the component designer.

There are many things to be said about these methods and especially what can and what cannot be done in them. We will see much of that later. But let us start with looking at the system's view, which is their abstract, external side. For it is the system that will invoke these methods and thus in a way determines what they can be used for.

The three methods are

`initialize()`
: This method is called when the system is initialised. Not surprisingly it is intended to initialise the component.

`execute()`
: For passive components this method simply contains the function or executional behaviour of the component. Once the system is running this is executed whenever the component is scheduled to run. For an event component this will be on the arrival of the associated event.

  For active and event components this method often has the additional task of taking part in implementing the synchronisation between the component and its environment (see `synchronize` right below).

`synchronize()`
: This method is not present in passive components. It is, usually in interaction with the `execute` method, used for synchronising an active or event component with its environment.

To the system an application simply looks like a tree of component instances, each having these two or three methods. And running the application is just a matter of invoking these meth-

ods in some, generally cyclic and often concurrent, order. It is a bit like playing a piece on a piano, where each key represents a component method, and pressing it invokes the function.

Note that as much as keys don't press other keys, components do not invoke other components' methods. To be more specific: a composite component does not invoke its subcomponents' methods. All behaviour invocations are one way or another done directly by the system.

5.3.2 Component Base Classes

The component base classes are generated from the CoCo component specification. Most of what happens inside these classes is considered to be private to the system and is made inaccessible for its subclass, the component class, i.e., user written code.

The primary task of the base classes is to implement the structural aspects of the component as they are found in the CoCo specification. Even though all this is hidden it is useful (especially when debugging or testing) to have some idea of what is going on. So let us take a glance.

The first thing to look at is the instantiation of components. If a component has subcomponents then instances of their classes will be incorporated in the component's base class. So an instance tree given in CoCo will result in an object tree in the generated code.

Figure 5.3 shows a CoCo specification of a simple component composition and the class diagram for its implementation. It is important to realise that component classes are being instan-

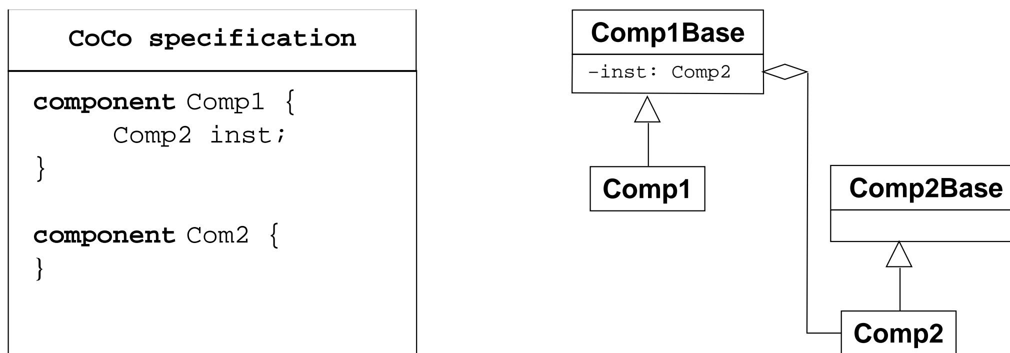

Figure 5.3: Nested Components with their Class Diagram

tiated but that it is done in the component base classes. And that these instantiations are made private to those base classes, effectively hiding the entire object tree for all behaviour code. This is done to avoid accidental (or intentional) abuse like directly accessing subcomponent's methods or data members.

Component base classes also partially implement ports and connectors (the remainder of the implementation is found in a DataStore class that we won't go into here). Again most of this is hidden. But ports need to be accessed from behaviour code and therefore the base class provides special methods[1].

Depending on its type the following methods are provided for each port:

`put_`*`port`*`(`*`type arg`*`)`
: Writes *arg* to *port* and thus all ports it is connected to. The type of *arg* is the value type of *port*. In case *type* is structured *arg* will be a reference (*type &arg*) to avoid needless copying. This method is provided for `output` and `inout` ports.

`get_`*`port`*`(`*`type *arg`*`)`
: Gets the last value that was written to any of the ports connected to *port* (or *port* itself if it is an `inout`) and copies it into whatever *arg* points to. The type of *arg* is the value type of *port*. This method is provided for `input` and `inout` ports.

[1] Having all this hide and seek around ports may suggest their implementation is inefficient. In fact the contrary is true. An access amounts to no more than one or two memory read/write operations. See [7] for details.

As we know from the model active and event components have their own data space that needs to be synchronised with the data space of their environment. This is of course only necessary for ”shared” data, so it is restricted to the component's ports. In the implementation this means that for every port value there exist two copies, one in the private data space of the active component and one in the data space outside. During synchronisation (execution of the `synchronize` method) these values can be brought up to date.

The base component class provides methods to perform these updates. Since in general you don't always want to update all values at the same time a copy method is provided for each port separately:

`export_`*`port`*`()`
: Copies the internal value of the port to the external value, making it available to the component's environment. This method is provided only for `output` and `inout` ports of active or event components.

`import_`*`port`*`()`
: Copies the external value of the port to the internal value, making it available to the component itself and its subcomponents. This method is provided only for `input` and `inout` ports of active or event components.

These methods do not return a value in any way, they only perform the copy. This means that to actually make a new port value visible to components on the outside you first have to call `put_`*`port`* to set the value and then `export_`*`port`* to copy it out. Similarly you have to call `import_`*`port`* before `get_`*`port`* to retrieve an external value.

According to the model these updates may only done while a component is synchronising. This means in practise that the `import` and `export` methods should only be called from a component's `synchronize` method[2]. And that passive component base classes do not have these methods.

Timer Components

Timer components are event components that have a special `Timer` property set. For example:

```
event component Timered {
  properties { Timer = "timer"; }
}
```

This property has two effects:

First, the component will be equipped with a timer (named `timer`). A timer is a sort of alarm that produces an event when it times out. Since a timer component is an event component this event will trigger its execution. Since the alarms go off with great precision, this allows execution of behaviour with very accurate timing.

But what makes timer components really special is that they can be controlled from the outside, by their parents. This is the only case in which components can communicate other than through ports. The reason for this is that communication through ports, as we just have seen, requires synchronisation, which cannot be done immediately. Making it impossible to do accurate timing, the reason what timers are for in the first place.

The access a parent gets is however indirect. It can only manipulate the timer and thereby triggering the execution of the component. So it is not really communication as in transferring data. It can only give it an precisely(!) delayed kick.

To manipulate the timer the following timer methods are provided to the parent:

`setAbolute_`*`timer`*`( long time )`
: Sets the time-out to an absolute time. Timer is started

[2] This is a restriction that unfortunately cannot be enforced.

`setRelative_`*`timer`*`( long time )`
Sets the time-out to a time relative to the current time. Timer is started.

`reset_`*`timer`*`()`
Resets and restarts a timer that has a relative time-out.

`cancel_`*`timer`*`()`
Disables the timer. Needs a new time-out to be set to be used again.

In the method names *`timer`* stands for the pattern *`inst_name`*, where *`inst`* is the name of the instance and *`name`* the name of the timer (as given in the `Timer` property)

It is possible to have more than one timer by specifying a list of space separated timer names for the property. You should remember though that they will all execute the same behaviour and that no means are provided to distinguish which timer went off.

5.3.3 Component classes

Component classes are the ones you have to provide (when using the IDE templates will be created automatically for you). What they need to contain is the component's behaviour. We can distinguish three kinds of behaviour: initialisation, function, and, if applicable, synchronisation. Behaviour is given by implementing the three abstract methods from the component type base class: `initialize`, `execute`, and `synchronize`.

Because component classes are specified in the implementation language there are very few restrictions on what you can do. This is intentional, so that you won't be unnecessarily limited by the system. But this freedom comes with a responsibility: there are some rules about things you shouldn't do. These rules make, if complied with, a component well-behaved. Which means that it can more easily be reused and interchanged with other components. Furthermore it will allow the application of verification and analysis methods that are being developed for PECOS. And finally there are also benefits in the area of debugging and testing. In short such a component will be much more useful.

Before we go into the details of the different kinds of behaviour let us look at some of those rules we should submit ourselves to:

Rules for Behaviour

Rule 1 Keep behaviour local

This most important rule relates to the essence of component based design. It was touched upon by the adjective "local" when we started talking about behaviour. It means that in principle a computation taking place in a component should only have effect within the component itself. It should not need to be aware of or influence its environment (the remainder of the system, including its subcomponents) other than through the means provided, i.e., it's ports.

Rule 2 Stay within the system

Staying within the system means not using your own ways of achieving something for which mechanisms are already provided by the model. For example:

- don't instantiate other components: use instances in CoCo
- don't start your own threads or processes: use active components
- don't try to implement your own scheduling using timer components
- don't communicate with other components for instance using global or class variables (`static` member variables in C++): use ports and connectors

In short, don't use any back-doors. Use the system as it is meant to be.

Rule 3 Avoid keeping the processor longer than necessary

As we will see later one of the biggest complications is to get all pieces of behaviour to be run in time. This is greatly complicated if components hold on longer to the processor longer than strictly necessary. Beware of unbounded loops, or indefinite blocking, or other ways of keeping other components from getting their go.

The system passes control to a component in good faith, under the assumption that it will get it back in due time. There is no preemption within a thread so the system has no way of taking back control from a component that doesn't finish.

In general it is good practise to keep urgent tasks as short as possible and run long calculations at a lower priority so they don't interfere too much.

Rule 4 Do not allocate heap memory

A final thing to look out for is that tight memory systems like field devices often have not much of a heap. Therefore memory allocation can easily overrun the limit and fail. To avoid any risk of this happening (quite disastrous in real-time systems) memory allocation should be avoided completely at run-time (i.e., directly or indirectly from `execute` or `synchronize` methods). This puts a ban on the use of new, malloc, and related functions. Of course, objects can still be created on the stack (not in Java, but the real-time extensions provide ways around this problem).

This rule can be loosened for initialisation. In principle it is not bad to allocate memory at that time. But especially if the `initialize` methods are also used to reset the system you should take care to reuse any memory allocated before.

For each of these rules probably exceptions can be found. In those special cases they may need to be broken. However, this should only be done after careful consideration of all options and finding that the feature cannot be implemented in any other, more appropriate way. Any such deviation should be marked and documented very well. And reported so that future versions of the system can provide a proper solution.

5.4 Component Behaviour

We distinguished three categories of behaviour. Here we will take a closer look at each one of them.

5.4.1 Initialisation

There are two opportunities to initialise a component: from within the constructor and from the `initialize` method.

The natural candidate for component initialisation is of course its constructor. However a problem is that at the time it is called it is uncertain how much of the remainder of the system is already in existence. So certain initialisations (like assigning port values) simply cannot be safely done. Furthermore, initialisation (as opposed to construction), is started under your control. So in principle this can also be used to bring a system completely back to its initial state without having to restart it. But of course this only works if the constructor is not used to set initial values. In short, the constructor should only be used to construct the component's parent class (the component base class) and any additional member variables it may have.

At the moment `initialize` gets called, everything is present and there are no restrictions anymore on what can be done (other than the rules mentioned before). So this is the preferred time to give member variables and ports their initial values. Of course, initialisation should follow the rules and therefore should do no more than local initialisations. So setting up member variables (other than instantiating) would be the main task.

<table>
<tr><th>Component in CoCo</th><th>Component Class in C++</th></tr>
<tr><td>

```
component Switch {
    input  byte select;
    input  long value;
    output long out1;
    output long out2;
}
```

</td><td>

```
class Switch: public SwitchBase {
    public Switch(int id): SwitchBase(id) {}

    void execute() {
       int select;
       long value;

       get_select( &select );
       get_value( &value );
       if (select == 0) {
           put_out1( value );
       } else {
           put_out2( value );
       }
    }
}
```

</td></tr>
</table>

Table 5.1: A switch component

Regarding setting port values you should be aware that if another component has a port connected to your port, it may override the value you set if it happens to be initialised after yours. In other words it depends on the order in which components are initialised.

This order depends on the component hierarchy and the order in which components are instantiated in the CoCo specification. The rule is that for a given composite component first the `initializes` of its subcomponents are called, in the order of instantiation in CoCo, and then its own. So this means that components are initialised depth-first and post order. The process starts at the top-level component (so its `initialize` will be called last).

You should also know that this process runs before any threads are running. So for initialisation no distinction is made between passive, active, or event components. The component hierarchy is strictly followed.

5.4.2 Execution

With execution behaviour we mean performing any computation other than synchronisation. So for instance for the Clock component from the introductory example it means getting the current time from the system and for a PID component it means computing a new control signal based on its desired state, measured inputs, and some stored, historical values. In general it entails reading input ports as needed, performing some calculation possibly with the use of local variables, and writing the results to one or more output ports.

Leaf (non-composite) components can be expected to always have some function (or else they have no purpose in the system and they might as well be left out). Composite components may or may not do anything. Often they will function purely as a connection box for their subcomponents.

The function of a component will always be located in the `execute` method of the component.

Table 5.1 shows an example of a simple switch component, that, depending on the value found on its `select` port, copies the value on its other input to one of its output ports. The CoCo specification and the component class are shown next to each other (only the relevant parts are shown).

5.4.3 Synchronisation

The following requires a good understanding of the execution model as described in Section 3.3. Some concepts will be explained again, but this time from a more implementation oriented point of view.

Active and event components run asynchronously from their environment, i.e., the composition they are part of. In practise the active components run usually in their own thread while the event components are executed in the high priority system thread. But whichever way they are run, to communicate with their environment they need to synchronise.

For synchronisation the special method `synchronize` is provided. This method will actually be run synchronously with the component's environment (its active parent), which allows it to freely access external port values without interference from sibling or parent components. It is important to understand that since the `execute` method runs within the active component's own thread, `synchronize` and `execute` are actually run asynchronously from each other.

Synchronisation within a component typically entails bringing the external and internal port values up to date. As we have seen this is done using the import and export methods. But you cannot use them arbitrarily since you are dealing with concurrency and data can be trashed if two threads write/write or even write/read simultaneously.

Ways to deal with this are many and it is beyond the scope of this chapter to go deeply into this. So here we will restrict ourselves to a few typical examples. But before we do so it is good to realise some features of the PECOS model that will make our life easier.

First of all, let us look at the port values of an active component. As we saw in Section 5.3.2 they have external and internal copies. The external ones are accessible only from the component's `synchronize` method that runs synchronously with the component's external world. So there is no problem there: everything is synchronous. For the internal ports values the same is true but with one exception: this same `synchronize` method has access to them as well. So we can conclude that synchronisation problems can only exist between the `synchronize` method and whatever is done inside the active component (through its own behaviour or that of its subcomponents). And also that possible conflicts are restricted to accessing (directly or through a connection) ports of the active component.

Furthermore since this `synchronize` method run synchronously with possible other `synchronize` methods in the external environment, this environment will never be involved in more than one synchronisation at a time. In other words there will never be three or more way synchronisations.

And finally, because of reasons of separation of concern you don't really want subcomponents to be aware of the fact that they need to synchronise just because they happen to be connected to a port of an active parent component. So in general all we need to focus on is synchronisation between the `synchronize` and the `execute` methods of one active component.

Having it narrowed down this far, now let us look at a few synchronisation examples. Please be aware that the examples are simplified and stripped of all code that is not essential for the explanation.

A simple case A simple and quite general case is where the two parties use locking to keep each other from interfering with each other.

Take for instance (see Figure 5.2) a highly simplified valve controller that has only a human interface (HMI), i.e. a display and keyboard through which it can be controlled. The control algorithm runs frequently, say every 20 ms, while the HMI runs only every 100 ms. The controller has a set-point, the desired state of the valve, measures the actual state, and tries through the actuator to get the two to match. The HMI displays both the desired and actual state and allows the user to key in a new value for the desired state. Running the control loop has absolute priority and should not be delayed by the HMI.

We can implement this by using an active component for the HMI and giving it the synchronisation mechanism as shown.

As you can see, the two methods use a Mutex (provided by the RTE) to ensure exclusive access to the ports. Once one side has acquired the Mutex a call of `acquire` on the other side

<table>
<tr><th>Component in CoCo</th><th>Component Class in C++</th></tr>
<tr><td>

```
active component HMI {
    input  float desired;
    input  float measured;
    output float newSetPoint;
}
```

</td><td>

```
class HMI: public HMIBase {
    private: boolean haveNewSetPoint;
    private: Mutex   lock;

    void execute() {
       float in1, in2, out;

       lock.acquire();
       get_desired &in1 );
       get_measured &in2 );
       lock.release();

       // display in1 and in2 and
       // possibly get new out value here

       if (haveNewSetPoint) {
          lock.acquire();
          put_newSetPoint( out );
          lock.release();
       }
    }

    void synchronize() {

       lock.acquire();
       if (haveNewSetPoint) {
          export_newSetPoint();
          haveNewSetPoint = false;
       }
       import_desired();
       import_measured();
       lock.release();
    }
}
```

</td></tr>
</table>

Table 5.2: A simple locking example

will block until `release` is called and vice versa.

In this particular example the locking times are very short so everything works fine. But in general a time critical component, like the controller, should not be blocked. Without loss of functionality it could simply skip the data exchange and perform it in the next cycle when it can expect free reign due to the difference in cycle time.

To allow this the Mutex should provide a non-blocking version of the `acquire` method. The current implementation unfortunately lacks this feature.

Triggering Action A second example is an implementation of a Non-Volatile RAM component, that is for instance used to write the state of your controller to, so it can restart where it left off.

Writing to this NVRAM takes very long and while this is taking place you generally cannot afford to halt the application. So a better implementation is to take a snapshot of the state, i.e., a copy of a consistent set of values, to a separate buffer, and then write the buffer contents to the NVRAM using some low priority background process.

The simplified component in Table 5.3 implements this behaviour. Its `execute` waits for `synchronize` to tell it that it can start to write. When told so, it does the writing and when done sets a flag (`busy`) to signal it is ready for another go. `Synchronize` on the other hand, won't accept outside requests to save the data as long as the writing is still going on. But if the component is ready it imports the data and sets off `execute`. To synchronise (wait, release) between the two methods a guard is used, a mechanism provided by the RTE.

With the outside world a simple flag protocol is used. To request data to be saved, the `request` port should be switched to `true` Once the component is done it will switch the flag back to `false` to indicate it is ready to accept new requests. Before that it will not look at the external value of the port and thus ignore any value changes. Since `synchronize` runs synchronously with the outside world no guards or similar machanisms are necessary[3].

No synchronisation is needed In some simple cases no special synchronisation is needed at all. A simple example can be found in the EventLoop component of Chapter 2. It only needs to export a single, 1-byte value. Since this an atomic action for the processor this cannot be preempted and there is no need for locking.

Note that this can only be done for simple, sufficiently small values, or, under very specific conditions, for a collection of them. Small is here related to the processor word size: even copying a 32-bit float or integer on a 16-bit processor may be a two instruction operation, that can actually be preempted.

5.5 Scheduling Behaviour

Now our components have behaviour it is time to turn our attention to how to actually make them run. The chapter on the model already prepared us for this.

We said before that an application can be seen as a collection of `execute` and `synchronize` methods. Let's call these bits of behaviour *action*s. According to the model every such action is associated with one specific active or event component and will run in that component's thread or event handler. So these components actually impose a partition, a separation into disjunct subsets, on this collection. And within each of these subsets actions are run in some sequence, synchronously with respect to each other.

The difference between active and event components is that the former are usually cyclic and run at varying priorities, while the latter are triggered and run at top priority. Because event components need to handle events quickly they will also rarely have more than one action.

Irrespective of this, however, what needs to be done is to define these sequences for every subset. There are two ways to do this. The first is indirectly, through specifying properties that contain timing constraints on components. A synthesis tool will then convert these into valid

[3] Or possible without violating Rule 2

Component in CoCo	Component Class in C++

<pre>active component NVRAM {
    inout bool request;
    input Data data;
}</pre>
	<pre>class NVRAM: public NVRAMBase {
	   private: PecosGuard guard;
	   private: boolean haveRequest;
	   private: Data buffer;
	
	   void execute() {
	      while (1) {
	         guard.wait(); // wait for release
	         get_data(&buffer);
	         // write buffer to NVRAM here
	         busy = false;
	      }
	   }
	
	   void synchronize() {
	
	      if (! busy) {
	         if (! haveRequest) {
	            import_request();
	            get_request(&haveRequest);
	
	            if (haveRequest) {
	               busy = true;
	               import_data();
	                 // now let execute() go
	               guard.release();
	            }
	         }
	         else {
	             // just finished a request
	             // set request flag to false
	             // to signal readiness for
	             // new requests
	            haveRequest = false;
	
	            put_request(haveRequest);
	            export_request();
	         }
	      }
	   }
	}</pre>

Table 5.3: A simplified NVRAM component

sequences and write them out as CoCo schedules (see 4.7). The other option is to write these schedules directly by hand. There is actually a third option as it would also be possible to do both: generate schedules first and then fine-tune them by hand. However, since at the time of this writing the tool was not available we will only look at manual scheduling here.

Writing schedules is maybe the most complicated step in getting an application to do what it is supposed to do. On one hand there is the problem of determining a proper sequence within a single thread. But then they also have to be made to work together, so that they don't block to long, produce answers in time, etc. And of course, there is also a dependency on what happens inside especially the synchronising actions.

Let's start with what we can do in schedules:

- specify actions in any order (independent of the composition hierarchy)
- invoke actions as often as you want (including never for instance for empty actions)
- insert waiting periods
- have the sequence repeated with a certain period or run only once

There are also things we cannot do:

- specify actions not belonging to the active or event component at hand
- change the schedule in time

So to go back to our piano playing metaphor: a schedule represents the part of the tune played by a single finger. Unlike normal piano play each finger is only allowed to play a specific set of keys (and no other finger can touch them) and has to play the same sequence over and over again. So for one finger the tune may be complicated but it is sequential (one key at a time) and repetitious.

To achieve a certain behaviour it is necessary to run actions in a specific order. A thing to keep in mind is that in cyclic schedules there is no real start or end. So for instance if there are two actions A and B of which we know that B needs to be executed after A, then in the schedule it is usually (depending on proper initialisation) fine to execute B first and then A. This because B will be executing after A in the next cycle so the requirement is still met. However like ordinary programs schedules should be written in an intuitive and easily understandable way. So for instance in the example above it would still be preferable to have A scheduled before B within the same cycle.

Another thing to keep in mind is that connectors may suggest a direct 1-cycle data-flow but the schedule may do differently. Take for example a pipeline of components, where the outputs of one component are connected to the inputs of the next. The connections suggest that the actions should be performed following the flow of data, so that the entire computation finished in one cycle. But there may be reasons to use the pipeline differently and for instance execute them in the opposite order. [4]. So clearly the schedule is in part defining a composition's behaviour.

Let's now look at some scheduling examples:

A simple schedule

Most composite active components need a schedule. A simple example is the top-level Device component from the example in Chapter 2. It simply schedules the actions in the order they need to be invoked. No delays between the actions are used.

[4] It is probably a good idea to make such deviation of the expected very obvious and its intended effect well documented

Free Running

Some components don't need a real schedule: they just need to be started, and then run their own infinite loop. A good example is the NVRAM component discussed in 5.3. This component is in a continuous loop that is suspended when it is waiting. Of course, keeping Rule 3 in mind, as we did in the example, to implement this waiting you should use RTE provided mechanisms not something like a polling loop. The latter would burn processor cycles for no use and probably get in the way of other tasks.

The schedule for the NVRAM component is extremely simple and looks like this:

```
schedule run_NVRAM of xxx.nvram at 1 /* low priority */ {
        exec at 0;
}
```

Leaving out the cycle time means that it is run only once, at the start of the application. The component has then to provides its own infinite loop.

5.5.1 Priority scheduling

Until now we have mentioned the word priority a couple of times but never explained what this would mean. Actually in our discussions we have more or less assumed that schedules or threads are run concurrently. Of course, this is a simplification. In reality all our schedules have to run on a single processor. Meaning that the schedules themselves need to be scheduled. And this is where the priorities play a key role.

Running the schedules is done by the RTE, usually by making use of the thread scheduler of the underlying RTOS. All control we have over this process is by assigning priorities. But before we can do this we need to know better how the RTE works. For that some standard scheduling terminology is required that we will introduce with only a rough explanation.

For our explanation every thread will always be in one of three states: running, ready, or waiting.

A thread is *waiting* when it is unable to run because it has run out of things to do (next action is not due yet) or it needs to wait for something to happen, like a lock being released.

A thread is *ready* if it is able to run, but not running because another thread already happens to be running.

A thread is *running* if the processor is actually executing its code. Obviously at any point in time there can be only one thread running.

Suspending a running thread to be able to run another thread that is ready is called *preempting* it. The preempted thread will move into the ready state.

So now let us see some rules according to which the RTE schedules threads:

- a running thread will only be preempted in favour of a higher priority thread
- when a thread goes into the wait state (because it blocks, sleeps, etc.) the highest priority thread that is ready will be run
- if more threads than one are ready and have the same priority one is picked (no assumptions should be made about which one[5])

The major point to remember is that the highest priority thread will always run first and exclusively until it finds a reason to wait. And lower priority threads can only get the processor if and for as long as all higher ones are waiting. This obviously makes another case for Rule 3 of component behaviour.

Of course this priority mechanism also affects the timing we specify in our action schedules. Apart from the highest priority schedule, we can no longer expect the specified times to be met exactly. For, if a higher priority thread happens to be running, there is no way an action can be run

[5] Usually threads are selected "round robin". Some RTOSes allow this to be configured.

in time. Of course, you could try to ensure the former cannot happen. But that requires finding a set of schedules that can never get in each others way, which would be impossible when using components like the NVRAM that have running times that can exceed the cycle time of other parts of the system.

The solution is to loosen our interpretation of the start times given in a schedule. Instead of seeing them as fixed start times they are now seen as advisory times. But in addition it will be required that actions finish before the next start time. So effectively the start times divide the schedule cycle into intervals in which the actions have to run. This ensures that actions will still be performed at the proper time if no higher priority task gets in the way. And that, even if an action gets delayed, it will be finished before the next action is due. This warrants that even though actions may run at varying times the schedule cycle itself will not be delayed, which would compromise the system. The RTE can actually be configured to verify that these deadlines are met.

None of this tells you how to write these schedules or assign priorities. Unfortunately there is no golden rules telling how to do this. But we can say a few general words on designing these schedules.

The main one is that the system should be designed in the most natural, intuitive way. It is only when you find that you can't get it to work that you should consider sacrificing the design. And keep in mind, even if it is wasteful in time and other resources, as long as it meets the design constraints, it works. It being an intuitively sound design is important, especially when doing maintenance, more than its ability to break the speed record.

But at times it may be necessary to change the design to be able to meet the deadlines. An example is in [6] where a component needed to wait for a response from the hardware after querying it. In the final design it proved not to be necessary to actually modify the component but if timing had been more tight, it would have been necessary to split the component into one part that sends the query and another that waits for the reply. After this separation the latter can be run in the background so processing can continue during the waiting period.

It is obvious that real-time critical parts of the application should be run at a high priority, so they can meet their strict deadlines. But on the other hand you want to avoid running too much action on a high priority, since that does not leave much opportunity for other tasks to do their job. So it is often good to isolate the part that has the hard real-time requirement from its supporting calculations and move the latter to another lower priority schedule. That way high priority tasks take the shortest time possible and also won't get in each others way.

5.6 Main and PecosDevice

So now we have our components and the schedules. We still need to get the system to run. For that we have to write a last little piece: the main routine. Here we have to invoke two class methods of a class we have briefly mentioned before: the PecosDevice class.

This class contains two important data structures. The top-level component class is instantiated here and it also contains the implementation of the schedules that have been translated from CoCo. Once more they are part of the system that you should need no direct access to and are therefore hidden.

As usual methods are provided to allow you to do what is necessary. In this case there are two that need to be invoked by you:

`initialize()`
: This starts the initialisation process of the component tree or, in other words, initialises all components.

`run()`
: This actually starts the running of the application. It should be called after `initialize`.

As we will see in the next chapter, `main` may contain more than these two calls. But the following is what minimally needs to be done to get things running:

```
void main()
{
        PecosDevice.initialize();
        PecosDevice.run();
}
```

5.7 Summary

In this chapter we have seen how components can be given behaviour. To make components more reusable and better maintainable this behaviour should comply with some general rules.

The most complicated task of creating PECOS applications is to make sure that the behaviour of the different components gets to run in time to produce the desired results. This requires a careful design of synchronisation, schedules, and priorities.

Table 5.4: Specifying component behaviour

1. A component's behavior is specified in a subclass of a generated class
2. There are three categories of behavior: initialization, functional, and synchronization
3. In general a component is free to do what it likes, except
    - The behavior should be local
    - A component should use the system, not try to defeat it
    - Control should not be kept any longer than necessary
    - No memory allocation should be done
4. Synchronization is implemented between the `execute` and `synchronize` methods of active and event components.
5. Schedules specify which behavior is run when
6. There is a schedule for each active component
7. Schedules are run at different priority levels
8. The intricate interaction between schedules, priorities, synchronization, and function make up the behavior of the device

Chapter 6

Application building and deployment

6.1 Application Structure

After discussing CoCo and the C++ language mapping in the previous chapters we are well prepared to develop a C++ based embedded device.

In the following sections the device functionality of our embedded device is presented. This allows us to specify the required components that we are going to develop. Figure 1.1 shows the embedded device example, a valve positioner, by which the development process of a field device will be explained. A valve positioner is a field device used to increase or decrease air pressure that acts upon an actuator.

The positioner controls a valve by a valve stem that is mounted at a membrane. This diaphragm reacts upon pressure and moves the valve stem. The field device positioner supervises the stem position until the valve reaches the required position steared through the instrument controller. Positioners are generally mounted on the side or top of their actuator. They are connected mechanically to the valve stem. Thus enables the device to compare the stem's actual position with the aspired one. A positioner encapsulates a type of air relay which is used to regulate the pressure that directly acts upon the stem.

The valve positioner is able to overcome hysteresis, packing box friction, and valve plug unbalance due to pressure drop. It guarantees exact positioning of the valve stem in accordance with the controller output. Variations in design of valves cause non-linear behavior in nature. The relationship between valve capacity and valve travel is known as flow characteristic of the valve. In order to fulfil the large variety of control application requirements it is necessary to compensate the valve's non-linearity. In conclusion the main task of the positioner is to control the valve position by the positioners pressure relay, the membrane, and the valve stem.

Figure 6.1 presents a high level component view of the valve positioner. The picture simplifies the device to stress the most important signal flow characteristics. The components have been developed from an existing architecture where we mapped function modules to components. In order to identify the relevant components of a new system those well-known methods for OOA (Object Oriented Analysis) such as use cases, context diagrams, event lists (Yourdon method), CRC cards, etc. can be used.

The current valve stem position is read using an `ADC` (Analog Digital Converter). The `Controller` component takes the actual position and the required position (Setpoint), calculates the actual position error and evaluates the new output value. The `PWM` (Pulse Width Modulation) component converts this output value in a pulse width modulated signal by which the pressure relay is driven. The user can set the valve position using the local HMI (Human Machine Interface) that consists of three components `HMIControl`, `HMIModel`, and `HMIView`. The HMI display itself is connected via the I2C bus to the devices micro controller. The `I2C` bus component serves as a

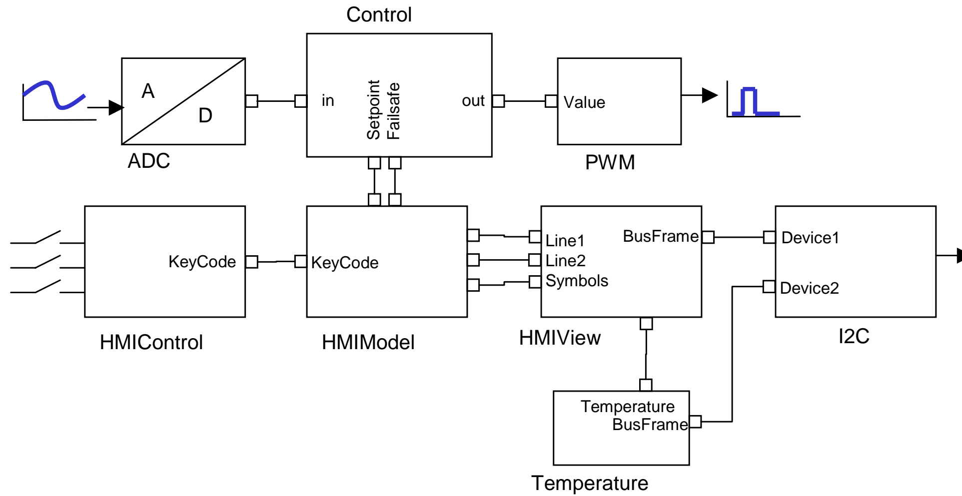

Figure 6.1: A simple valve positioner

hardware abstraction for the bus. In addition the environment temperature can be read from a sensor connected to this bus. It is used to control the display contrast.

6.2 Component and Device Specification

The upcoming sections will give you a more detailed view in the application. One of the most important issues for building component-based applications is to address the most critical real-time part of the embedded device first, because all other behaviors are dependent on it. Therefore we will now deal with the control loop of the positioner, by reason of its real-time requirements it is the most critical element in our embedded device.

As you can see in figure 6.2 crux of the matter is the `Controller` component. From this point of view the `ADC` and `PWM` component are part of `Controller`, the explanation for this will be given subsequently. The stimulation component was added to set the inputs of the `Controller` as long as no HMI is available.

This composition shows the data flow between the components for closed-loop control system. All data connections, which are not used for the control loop, such as device parameters, have been omitted. The arrows pointing from and to `YOUT` and `PosADC` are indicating the connection to the physical process.

The composition: `YOUT` and `PosADC` serve as hardware abstractions. `PosADC` provides the absolute position of the valve stem. `YOUT` creates a pulse width modulated signal which is proportional to the air pressure affecting the stem position. The absolute position has to be linearized using the offset and the actual mechanical range to a logical position in the operational range. This linearization algorithm is implemented in `Position`. After linearization the `ControlAlgorithm` component takes the linearized current position and the setpoint input from `SetpointCalculation` component and calculates the new output value. The `SetpointCalculation` linearizes and converts the given setpoint to the logical and operational area of the valve. The control algorithm implemented in `ControlAlgorithm` consists of a structured controller coupled with zoning controllers. The `Shutoff` component steers in dependency of the current shutoff state the output to `YOUT`. In *shutoff mode* the output is set to `open` or `close`, dependent on the configuration by the user. In *normal mode* the output will be just looped through.

Table 6.1 lists all components including a short description of task and timing requirements. The real-time requirements are derived from the control theory the control algorithms are based

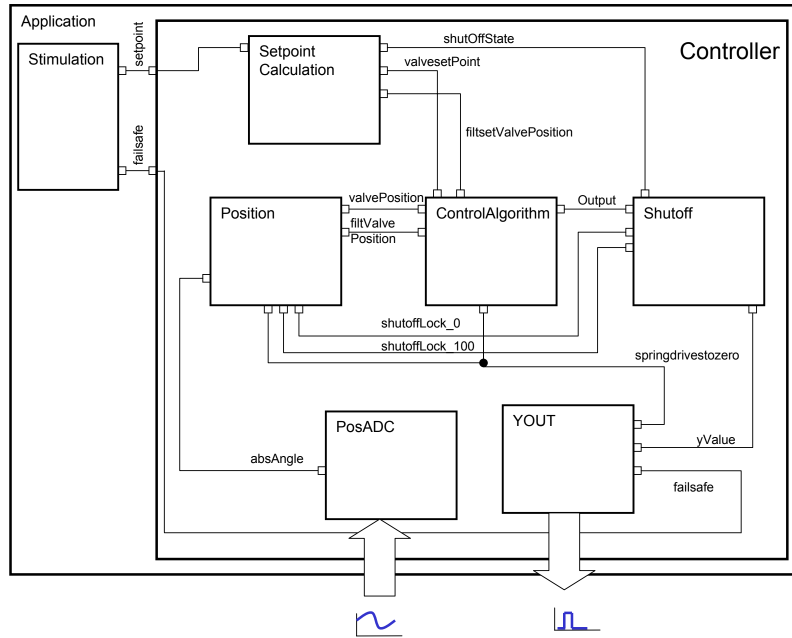

Figure 6.2: A very simple valve positioner.

on, or have its origin in user requirements such as the response time.

Now we have basically identified the components, the composition, and the data flow of our device. You may ask how the timing requirements are reflected in the specifications. This will be discussed in the next paragraphs.

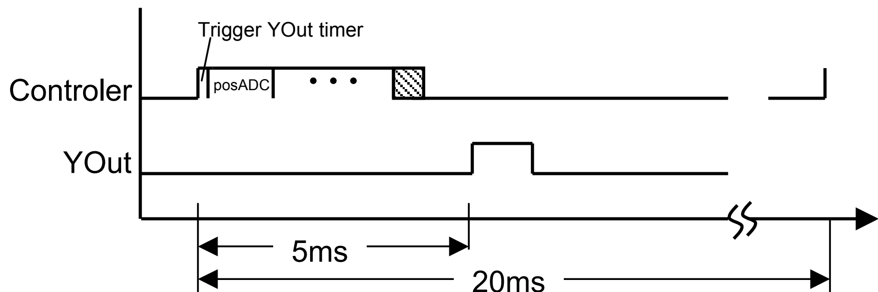

Figure 6.3: Required executing order and timing of our device

Figure 6.3 presents the required real-time behavior and the schedule as specified in table 6.1. There is a timing dependency between `PosADC` and `YOUT`. Therefore `YOUT` is specified as an event component with a timer. We can trigger this timer just before executing `PosADC`. The timer timeout is chosen in such a manner, that first the execution of `Shutoff` has finished and the new out value `yValue` is available and second `YOUT` can be synchronized before the time

Name	Task	Timing	Hardware dependency
PosADC	Gets the current absolute position of the stem from the analog digital converter	every 20ms	yes
Position	Linearizes and maps the absolute position to logical and operational area	every 20ms	no
ControlAlgorithm	Control algorithm with structured and zoning controllers	every 20ms	no
Shutoff	Switches between output of the control algorithm or shutoff value dependent on shutoff enable	every 20ms	no
YOUT	Creates a PWM signal with pulse width given as input	5ms after the PosADC has started conversion	yes
SetpointCalculation	Maps set-point 0..100% to operational area, set-point correlates to flow rate	1000ms	no
Stimulation	Stimulates the Controller component to influence the set-point and the shutoff	1000ms	no

Table 6.1: Components and their timing requirements of our device as found in discussions with the control experts.

expires. After time has run down `YOUT::execute` takes the new `yValue` from its input and sets the new PWM output. As a result of our composition we have achieved that the timing requirements match.

Now we can go on with building the components. Therefore two steps are necessary: a) specify the component in CoCo and b) implement the behavior. As an example we show the `Controller` and the `YOUT` components in detail. First of all, we have to set up a new ECLIPSE project. In section 2.3 the handling of ECLIPSE is explained. This time we create a `PECOS C++ Project` to use C++. The next step is to add a new C++ component `Controller` and enter its specification in the CoCo language. Figure 6.4 presents the ECLIPSE workspace after adding all components from table 6.1.

As we have discussed in the previous section the `Controller` component looks like:

```
active component Controller {
        Position            pos;
        ControlAlgorithm    contrAlg;
        SetpointCalculation setpCalc;
        Shutoff             shutOff;
        PosADC              posADC;
        YOUT                yout;

        input char failSafeEnable;
        input int setpoint;

        // Connectors follows here. Left out for clarity.
}
```

Here follows the schedule in CoCo for the active `Controller` component and for the event

component YOUT as derived from the discussion above:

```
schedule controlScheduler of Application.controller every 20 at 200 {
    {
        exec; // self, trigger timer of yout
        exec posADC
        exec position;
        exec contrAlg;
        exec shutOff;
        sync yOut;
        exec setpCalc;
    }   at 0;
}

event component YOUT{
    properties{
        Timer = "timer";
    }

    input char failsafe;
    input int  yValue;
    input char springdrivestozero
}
```

The execution order as specified in the controlScheduler was defined from the control expert. The Controller should be executed every 20 ms at high priority. Now the specification of our components and of our schedule is finished and we are able to build the complete project in ECLIPSE that creates the schedule and the component skeletons.

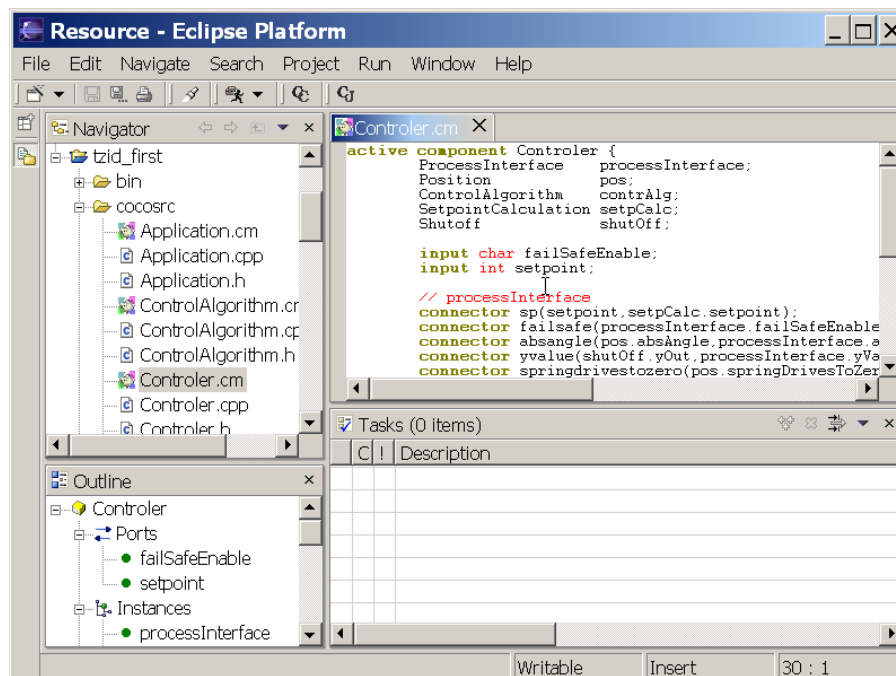

Figure 6.4: Eclipse development environment with opened valve controller project

The presented solution assumes that the cycle time of the `Controller` has a very low jitter ($\leq$ 100us). During design and development of the system special care must be taken to keep especially IRQ handler routines as short as possible. There are ongoing activities to extend the RTE to support these special real-time requirements.

Next step is to add and implement the behavior in C++ code to get a fully functional application that realizes the specified data flow and control flow:

- Implement the `execute` method of the `Controller`. Its function is to trigger the timer of component `YOUT` at the beginning of each cycle.
- Call get*, put*, import* and export* methods where required. Be careful with the different synchronize methods.

The source code of the `Controller` component consists of three methods in which the active behavior is implemented. Method `execute` triggers `YOUT's` timer, which was automatically generated by the codegenerator, see section 5. 2 for details on timers. In `synchronize` we import the two input ports of the `Controller` from the surrounding world. The `puts` commands are inserted for debugging.

```
#include <Controller.h> #include <stdio.h> // puts

void Controller::initialize() {
    // Add initialization code here
}

void Controller::execute() {
    // Add the component behavior here
    puts("Controller::execute() -> trigger yout timer");
    setRelative_yout_timer(5);
}

void Controller::synchronize() {
    // Add synchronization code here
    puts("Controller::synchronize");
    import_failSafeEnable();
    import_setpoint();
}
```

Now we are able to add the main functionality to our components step by step and compile and build our application for the target system. This is part of the following section.

6.3 Embedded Development Environment, M16C and embOS

In ECLIPSE [4] we specified the component descriptions and generated the C++ device skeletons and the schedule. Now we leave ECLIPSE and switch to an embedded development environment. In our example we decided to use Taskings EDE. Other developers may use the upcoming IAR C++ compiler environment or any other environment they are familiar with. `Tasking EDE` offers automatic makefile generation, debugger integration (CrossView or PD30), and automatic generation of hardware dependent definitions for your target. Other EDEs, such as `IARs Embedded Workbench`, can be used in the same manner. So, first the EDE has to be configured to set-up the whole project including the fitting makefiles, the target and the debugging monitor settings.

We used the Mitsubishi M16C 3-Diamonds Board as common target for the first development steps. Many field devices are based on this reference design. Mitsubishis board includes in- and outputs, such as eight LEDs, a controllable AD-converter and three buttons to trigger events by hand. Before the first usage the Mitsubishi ROM monitor has to be flashed to enable debugging connection to the host system.

The ROM Monitor Version 1 can be found on the CD-ROM of the 3-Diamonds Board set. In the board manual you can read further explanations how to flash the monitor.

The Tasking Embedded Development Environment offers several possibilities to configure your target. First you have to create a new workspace for your new project. The most important project options to use the M16C target are:

- Internal ROM 0xC0000, 256K
- Internal RAM 0x400, 20k
- Special Function Registers 0x00000, 0x400
- Enable Generate Start-up Code
- Enable and initialize PM1
- Use small memory model
- Set OS_LIBMODE_R for the C++ Compiler to support embOS

Reserved areas of the M16C without external memory should be:

- 0xFC000,0xFFFFF
- 0x0,0x400
- 0x4CCC,0x53FF
- 0x5FF,0xC0000

Don't forget to specify the right stack size e.g. 0x100 and address like 0x44cc and for the heap 0x200 and 0x3400 as well. More setting details regarding the debugger Crossview Pro are given in 6.7. Now add the Run-time Environment library `RTE.a` for the M16C and the embOS system `rtosNR.a` to your project. The correct path added to the include file path of the RTE lets include the environment headers to the project. Initialisation of embOS is located in `RTOS.h` and `RTOSInit.c`. These files have to be included from embOSs `INC` directory in your project tree. `RTOSInit.c` must be renamed to `RTOSInit.cpp` for C++ compatibility. The peripheral variable vector table has to be added in the `vects32.src` modified for our target:

```
ROM_MONITOR_VERSION equ 1

DEFSECT ".vecttab", FDATA, ROMDATA, MAX
SECT    ".vecttab", RESET

OFFSET  19 * 4
IF ROM_MONITOR_VERSION == 2
        DL      0xFF900    ; used by the Mitsubishi 3-Diamonds/Glyn ROM
        DL      0xFF900    ; used by the Mitsubishi 3-Diamonds ROM
    ELSE
        DL      0xFCB6B    ; used by the Mitsubishi 3-Diamonds/Glyn ROM
        DL      0xFCB6B    ; used by the Mitsubishi 3-Diamonds ROM
    ENDIF
END
```

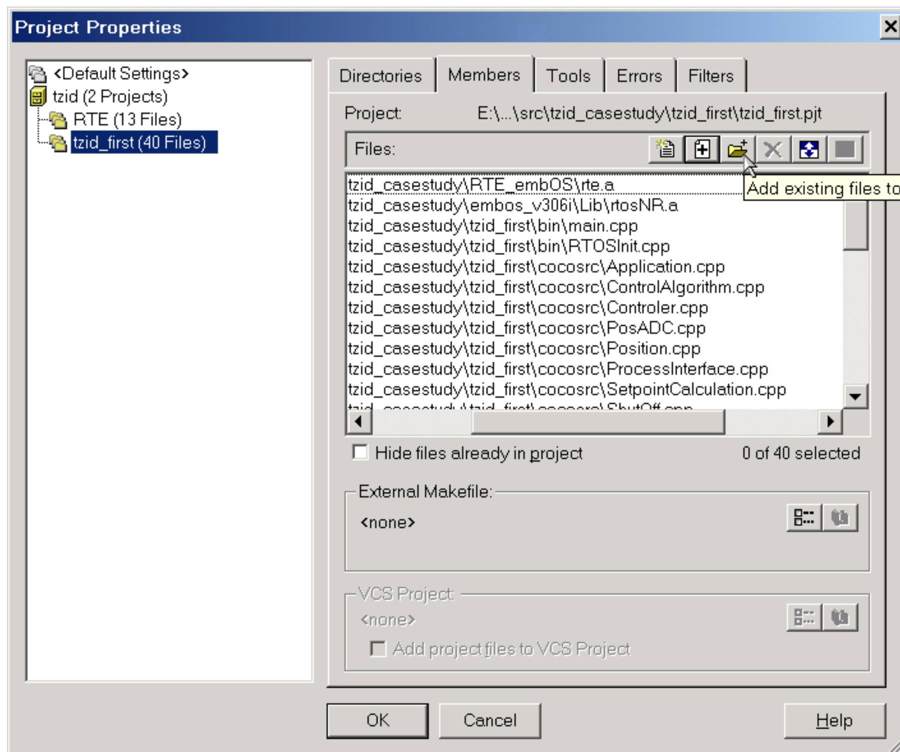

Figure 6.5: Adding generated files

Now you can include all the files, which are created by ECLIPSE to your project. Using the project properties from the project menu you are able to add files by extension, add all `.h` and `.cpp` files to your project, see figure 6.5.

After this set-up your project should include

- cpp and header files, which are generated in ECLIPSE
- `vects32.src`, peripheral variable vector table
- `RTOSInit.cpp`, initialisation and main file of embOS
- `RTOS.h`, definition file of embOS
- `rtosNR.a`, embOS library
- `rte.a`, run-time environment for M16C

The complete set-up looks like figure 6.6.

6.4 Implementing Main

A typical main implementation consists of initialising the operating system and the PecosDevice component:

```
#include "RTOS.h"
#include "PecosDevice.h"

int main(int) {
    OS_InitKern();         /* initialise OS */
```

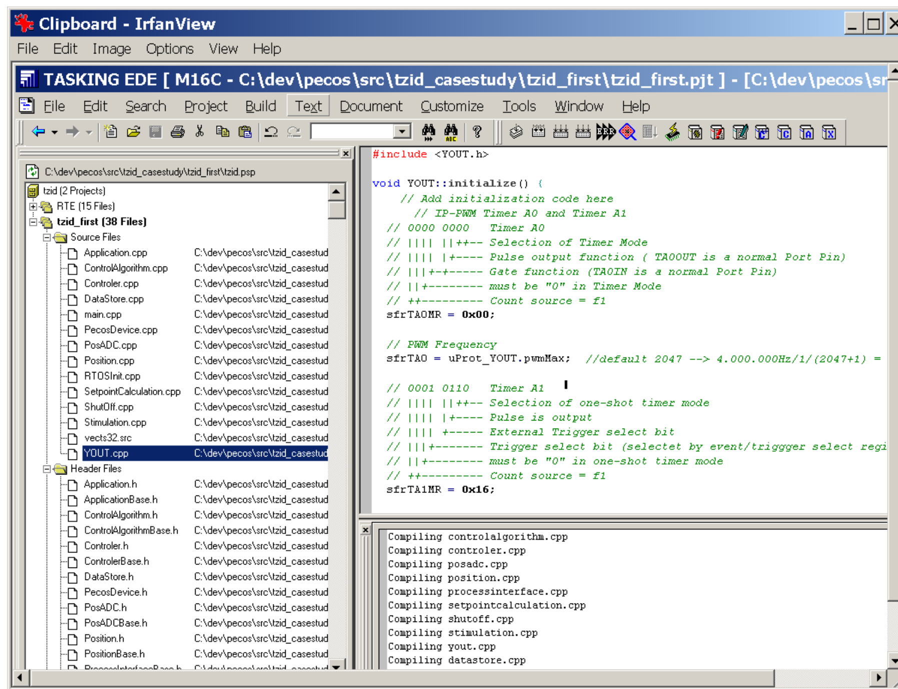

Figure 6.6: TaskingWorkspace

```
    OS_InitHW();              /* initialise Hardware for OS */

    PecosDevice::initialize(); // initialise Device
    PecosDevice::start();      // starts Device

    OS_Start();               /* start Operating Systems */
    return 0;
}
```

The embOS operating system is started with `OS_Start`, in which the main loop of the system is called. As the listed main routine shows the developer has not to implement more than the OS initialization.

6.5 Implementing Component Behavior

Now, as discussed in chapter 5 we can develop the behavior of the components. In the derived classes of the generated base classes code can be added to the `initialize`, `execute` and maybe `synchronize` method. To change ports or connectors or to create new components the ECLIPSE environment has to be used again. The Tasking EDE updates the modified files automatically, only newly created files have to be added to the project by hand.

6.6 Makefile and Building

After the configuration of your project set-up the EDE generates automatically a makefile `projectname.mak` for you. It is updated each time the project settings are changed. If you like to use your settings in other projects you can save and load them in the project menu. The EDE starts the make and build process by execution of the make command as shown in figure 6.7 and generates the binary executable in IEEE-695 format for our target.

Figure 6.7: Make

6.7 Debugging and Deplyoing

The debugging of PECOS applications only requires a source level debugger. Today, the only possibility to debug C++ applications is using a serial connection to the M16C system. Currently the ICE (In-Circuit-Emulator) PD30 is not able to de-mangle C++ sources, but Mitsubishi works on C++ support. Therefore we use the CrossView Pro Debugger 6.8 and Mitsubishis ROM Monitor Version 1 to debug applications in flash-rom. A serial cable connects the development system to the target system for remote debugging. The free running mode must be enabled to debug multi-threaded and timed applications. The M16C offers two hardware breakpoints. The pre-configured breakpoints at main and at SIO (Simulated In- and Output) should be disabled to make them usable in other code fragments. During a debug run of the application the stop command connects the ROM monitor to CrossView and shows the current code at the hit breakpoint. The debugging

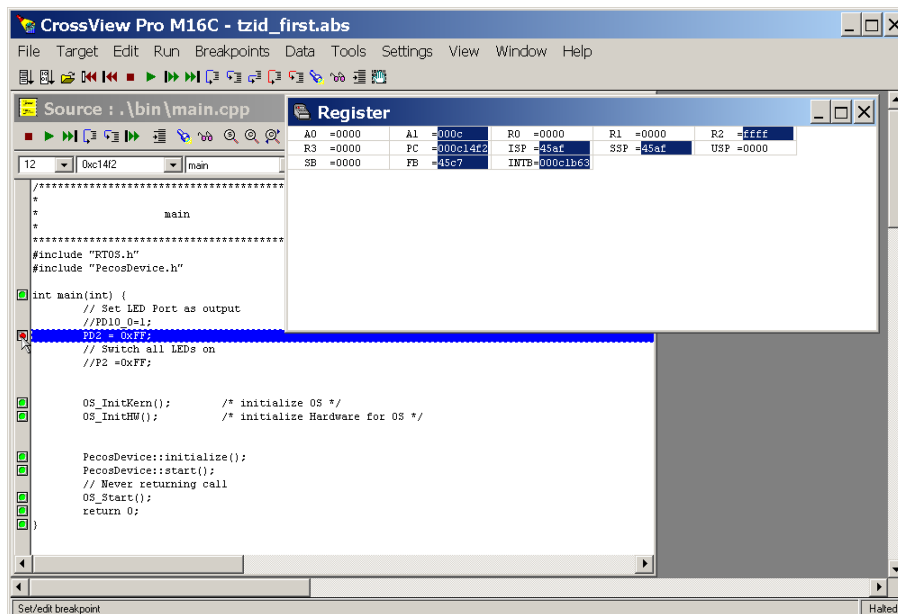

Figure 6.8: CrossView Debugger

6.8 Summary

This chapter discussed C++ development and the available tools and technologies. We have seen basic concepts of solving difficult real-time behavior and how easy it is to start building C++ embedded real-time applications step by step. In particular this chapter presented the use of CoCo, an embedded development environment and the M16C (see also Table **??**).

Table 6.2: Steps building embedded C++ application with real-time behavior

1. Create a PECOS C++ Project in ECLIPSE.
2. Identify critical timing issues.
3. Build your component specification in CoCo (`component.cm`).
4. Build the project (creates the base and utility classes).
5. Setup a project in your EDE.
6. Configure your project settings for your target.
7. Add the generated files from ECLIPSE.
8. Add the RTE and the OS.
9. Extend the main to the needs of your OS and the `PecosDevice` class.
10. Implement the behavior in C++(`component.cpp`) using your EDE.
11. Build the project.
12. Use your debugger for step by step debugging.

Appendix A

CoCo Grammer

This appendix shows the entire CoCo Syntax in extended BNF. Please not the meaning of the following special characters:

| : items separated by | are alternatives (exclusive or)

* : means that an item can appear any number of times

+ : means that an item has to appear at least one time

() : in addition, parenthesis are used to group items

Terminal symbols are written in capital letters. The characters "{", "}", "[", "]" and ";" are also terminal symbols.

```
cocoFile
        :   compilationUnits

compilationUnits
        :   (dataTypeDecl | component | task | propertySet)*

component
        :   plainComponent | abstractComponent

plainComponent
        :   componentType COMPONENT identifier isList hasPropertySets
            componentBody

abstractComponent
        :   ABSTRACT COMPONENT identifier isList hasPropertySets
            abstractComponentBody

componentType
        :   PASSIVE | ACTIVE | EVENT |

isList
        :   (IS abstractComponentNames | )

hasPropertySets
        :   (HAS propertySetNames | )
```

```
componentBody
        :   { (propertySettings | )
              (portDecl | instance | connector | ;)*
            }

abstractComponentBody
        :   {
            (propertySettings | )
            (portDecl | instance | role | connector | ;)*
            }

abstractComponentNames
        :   abstractComponentName (COMMA abstractComponentName)*

abstractComponentName
        :   Identifier

propertySettings returns
        :   PROPERTIES { (propertyStatement | ;)* }

componentName
        :   Identifier

portDecl
        :   portOption portType dataType Identifier hasPropertySets portBody

portOption
        :   OPTIONAL
        |   MANDATORY
        |

portType
        :   IN
        |   OUT
        |   INOUT

dataType
        :   (typeName | baseType) ([ Integer ])*

portBody
        :   { (propertyStatement)* }
        |   ;

propertyStatement
        :   IF String propertyStatement
        |   compoundPropStatement
        |   propertyInList

portAccess
        :   (instName DOT | )  portName

instName
        :   Identifier
```

```
portName
        :   Identifier

partialAccess
        :   [ Integer ]
        |   DOT Identifier

instance
        :   componentName Identifier hasRole hasPropertySets instanceBody

hasRole
        :   (IS roleName | )

instanceBody
        :   { (propertyStatement)* }
        |   ;

roleName
        :   Identifier

connector
        :   CONNECTOR Identifier hasPropertySets
            L_PAREN portAccessInList (COMMA portAccessInList)* R_PAREN
            connectorBody

portAccessInList
        :   portAccess

connectorBody
        :   { (propertyStatement)* }
        |   ;

connectorName
        :   Identifier

role
        :   ROLE componentName Identifier ;

task
        :   SCHEDULE Identifier OF instDesignator
            (EXTENDS scheduleName | cycleTime AT Integer)
            hasPropertySets
            taskBody
        |   SCHEDULE OF instDesignator eventTaskBody

instDesignator
        :   instName (DOT instName)*

scheduleName
        :   Identifier

cycleTime
        :   EVERY Integer
        |
```

```
taskBody
        :   {
            (propertySettings | )  (jobSchedInList | jobDefInList)*
            }

eventTaskBody
        :   { activity ( ; activity )* }

jobSchedInList
        :   jobScheduled

jobDefInList
        :   jobDefinition

activity
        :   SYNC instDesignator |   EXEC (instDesignator | )

jobDefinition
        :   Identifier ASSIGN activities ;

activities
        :   (activity | { (activity ;)+ })

jobScheduled
        :   (jobName | activities) AT Integer ;

jobName
        :   Identifier

dataTypeDecl
        :   TYPE Identifier
            (  (EXTENDS typeName | ) hasPropertySets
               { (fieldInList | propertySettings)+ }
            )
        |   IS dataType hasPropertySets
            (; | { (propertyStatement)+ })

typeName
        :   Identifier

fieldInList
        :   field

field
        :   dataType Identifier ;

baseType
        :   INT | BYTE | SHORT | LONG | FLOAT | DOUBLE | BOOL | CHAR

propertySet
        :   propSetType PROPERTIES Identifier
            { (propSetMember)* }
```

```
propSetType
        :   COMPONENT | PORT | TYPE | SCHEDULE |

propSetMember
        :   propertyStatement | propSetRefInList

propSetRefInList
        :   propSetRef

propSetRef
        :   PROPERTIES propSetName ;

propSetName
        :   Identifier

compoundPropStatement
        :   { (propertyStatement | ;)* }

propertyInList
        :   property

property
        :   propDesignator ASSIGN
            (MANDATORY | constExpression | compoundPropStatement) ;

propDesignator
        :   Identifier (DOT Identifier)*

constExpression
        :   MINUS (Integer | Float)
        |   Integer | Float | Char | String | TRUE | FALSE

propertySetNames
        :   propSetNameRef (COMMA propSetNameRef)*

propSetNameRef
        :   propSetName

propName
        :   Identifier
```

Bibliography

[1] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison Wesely, third edition, 2001.

[2] Alexander Christoph, Thomas Genssler, and Michael Winter. Description of the coco architectural description language and composition environment. Technical Report Deliverable D225.06, Pecos, 2001.

[3] Carolyn MacLeod and Steve Northover. Swt: The standard widget toolkit - part 1 and 2. http://www.eclipse.org/articles/index.html.

[4] OTI. Eclipse project home page. www.eclipse.org/.

[5] Bastiaan Schönhage. Pecos Run-time Environment C++ and Java. Technical Report Deliverable D4.6-2, Pecos, 2001. www.pecos-project.org.

[6] Andreas Stelter and Peter Müller. Field Device Component Specification. Technical Report Deliverable D1.4, Pecos, 2001. www.pecos-project.org.

[7] Reinier van den Born and Bastiaan Schönhage. Model mapping to C++ or Java-based ultra-light environment. Technical Report Deliverable D2.2.9-2, Pecos, 2001. www.pecos-project.org.