Deliverable D2.2.9-2

Model mapping to C++ or Java-based ultra-light environment

1. Identification

Project Id:	IST-1999-20398 PECOS
Deliverable Id:	D2.2.9 Model mapping to C++ or Java-based ultra-light environment
Date for delivery:	May 15, 2002 (revision 2)
Planned date for delivery:	May 15, 2002 (revision 2)
Classification	Public
WP(s) contributing to:	WP 2
Author(s):	Bastiaan Schönhage, Reinier van den Born, OTI

1.1 Abstract

This deliverable describes the language mapping for the Pecos Model. It defines how modeling constructs, such as components, ports and connectors, can be mapped onto an object-oriented target language, how the execution model can be implemented and how the resulting code fits the Pecos Runtime Environment. The purpose of this definition is the construction of an automated tool that translates designs specified in the CoCo modeling language to target language modules which can then be merged with the runtime environment and built into a running executable for the target platform. A proof-of-concept implementation covering the essential problems is well underway.
The design of the language mapping is subject to two main considerations: the target code has to be efficient and it needs to be robust. Efficiency is achieved by carefully choosing the target language features to be used and robustness by using access restrictions commonly available in OO-languages.
To support debugging, testing, and performance measurements variations on the mapping will be made.

1.2 Living document

Currently, this document is not finished. It describes a complete mapping of the Pecos model to the target languages C++ and Java except for the following:

- The mapping of timing properties to a feasible schedule. As this problem is closely related to timing verification it will either be described in that context or in a separate document.
- Support for component and (sub)system testing and performance measurements. This is future work.
- Support for partial connections. A rough outline of what is needed will be given.
- Support for model extensions like abstract components.

This document relies heavily on:

- D2.2.8-5 (the Pecos model [Model])
- D2.2.5-1 (CoCo description [CoCo])
- D4.6-1 (Pecos Runtime Environment [RTE])

1.3 Keywords

Pecos Model, Pecos Execution Environment, C++, Java, language mapping, code generation, scheduling, Data Store

1.4 Version history

Ver	*Date*	*Editor(s)*	*Status & Notes*
1.0	1/10/2001	Schönhage	First draft version.
1.1	8/10/2001	Schönhage	Updated after reviews from Pecos partners
1.2	11/10/2001	Schönhage	Changed into a living document with revisions Additionally, this document now contains both the C++ and Java mapping
1.3	11/10/2001	Schönhage	Changed from 2.2.3/4 to 2.2.9
2.0	20/04/2002	van den Born	First draft of second version.
2.1	15/05/2002	van den Born	Update after reviews from Pecos partners

1.5 Classification

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

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

1.6 Disclaimer

The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

2. Table of Contents

3. Introduction

One of the goals of the Pecos project is to build a system that allows designers to model their application as a collection of components and that provides tools to convert this model into a running executable. Such conversion tools are actually implementations of a language mapping: a mapping from the abstract modeling language to a programming language like C++ or Java.

A complete Pecos application specification is actually a hybrid one. Central is the model of the components, described in terms of their interfaces, their composition and their non-functional requirements (see [Model]). This model is specified using a specially developed component description language, CoCo [CoCo]. In addition, since neither the model nor CoCo covers this, the internal behavior of components needs to be specified. It was chosen to have the designer do this in the implementation language directly.

The language mapping actually specifies how the CoCo component descriptions can be translated to the target language. The code that the mapping produces together with the user provided behavior description and the Pecos runtime environment forms actually the entire application, that can be compiled and run on the target platform.

Special attention needs to be given to two aspects of the mapping. Of primary concern is the efficiency of the resulting code, both in time and resource consumption. This is a hard requirement stemming from the Pecos application domain. But an additional consideration is the robustness[1] of the system being built. The latter goes beyond the obvious requirement that component model should be mapped correctly. The fact that the designer specifies part of the system in the target language opens the possibility of, possibly unintended, unauthorized interference with the generated code, which can invalidate the model. Although this cannot be prevented entirely some measures can and thus should be taken to shield the generated part from abuse.

In addition, different modes of code generation need to be looked at. Having the general code generation in place gives opportunities for the generation of multiple variants of code for a single model. For example, during the development, systems may be generated that provide special support for debugging or profiling by inserting special statements or creating testing harnasses. Once the design is finalized and ready to go to production all excess code can be stripped and a more efficient bare bones implementation can be constructed.

3.1 This document

This document describes a language mapping suitable for OO languages supplemented with detailed instructions for Java and C++. More specifically, it describes:

- How modeling constructs, such as components, ports, and connectors, are mapped to constructs in the implementation language.
- How component interaction can be implemented.
- How the translated model will interact with the separately specified behavior.
- How all that code can be made to run with the use of the Ultra Light Runtime Environment [RTE].

Although not necessary to understand the general ideas of what follows, for the details of code generation the reader is expected to have a good, working knowledge of both Java and C++ (see [C++, Java]). Concepts from these languages, often referred to using keywords marked in **bold** face, will be used without further explanation. In the code samples used as illustrations mostly Java will be used as it is (slightly) more abstract than C++. When necessary the differences for C++ will be noted.

As said two special aspects need to be taken in consideration regarding language mapping: efficiency and robustness. Paragraphs dedicated to these issues will be marked with ⚐ and ☠, respectively.

This document is organized as follows. Section 4 will give an overview of the architecture of a complete application and show how the different pieces fit together. Then in section 5 the mapping of the structural

[1] Robustness of software can be roughly defined as being resistant to the introduction of bugs. See any handbook on sofware engineering for more on this.

aspects of the model will be given; followed in section 6 by a mapping for the execution model. Section 7 covers remaining issues that did not really fit any of the preceding sections. Section 8 will look at issues related to special purpose code generation, like for debugging and profiling. Finally, section 9 will give a summary of what has been accomplished. A complete example with code generated in C++ can be found in Appendix A.

4. Language mapping overview

A complete application design is mapped to a collection of related classes (see Figure 1) that can be divided into three layers: the runtime environment (RTE), the generated classes, and user provided classes.

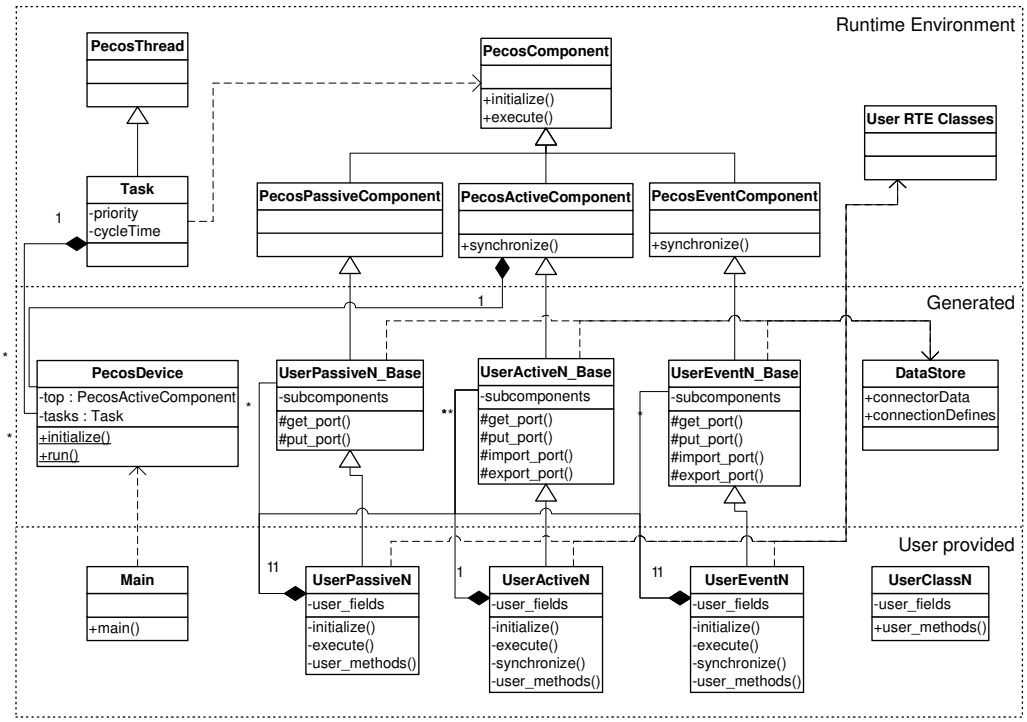


Figure 1: Component class hierarchy

The runtime environment is described extensively in [RTE]. For this discussion it suffices to know that it provides abstractions from the underlying operating system needed to run tasks, synchronize them, etc. In addition it provides base classes for each of the different component types (passive, active, event). These base classes serve as generalisations for the specific components of a design and, being abstract, enforce the presence of certain methods (`initialize`, `execute`, and `synchronize`) in the final component classes.

The component model is mapped to a number of classes that are collectively called the *generated code*. For each component in the design a subclass of one of the component base classes is generated, extended with references to its subcomponents and methods to access the component's ports. Furthermore, active and event components will be given methods for use to synchronize (import and export) values on their ports. In the diagram these classes are shown as the *ComponentName*_Base classes.
In addition two special classes are generated: the *DataStore*, which implements all connections in the design, and the *PecosDevice*, a class that instantiates the top-level component, holds definitions of all schedules, and provides methods to initialize and run the system.

The component classes that are generated only implement the structural aspects of the components. However, components can also have behavior. Especially leaf components, but others as well, may perform some calculations using and providing values on their ports by means of their `execute` method. Furthermore active and event components need a `synchronize` method to catch up with their environment. And often this behavior requires some initialization to run correctly which is done in the `initialize` method.
For various reasons the most elegant and flexible method to allow this behavior to be added to the system was found to be by requiring the designer to provide a subclass for every generated component class. There the above mentioned methods can be implemented. At the same time this allows methods to be overridden, members to be added, and external code to be included and used at will. Using subclassing finally provides a clear separation between generated and user provided code which allows a better protecting of the generated code. Further details regarding these "user" classes will be discussed in section 7.3.

In summary, a component specification in the Pecos development environment consists always of two parts. The first parts is the component description in CoCo and the second is the class specification in the target language providing the behavior.

A whole application specification consists of a collection of these component specification pairs and a little additional code that will be discussed in section 7. After code generation the whole application will fit the diagram given above. Together this forms the executable.

5. Mapping the structural model

This section describes how the structural aspects of the model, that is, the component instantiation hierarchy, ports, and connections, can be mapped to the target language.

5.1 Components

The main constituents of the Pecos model are components. Components are functional entities that are scheduled to perform some function and use ports to communicate. Components from the Pecos model are mapped to classes in the target language. Instantiations or instances of components are mapped to instances of classes, i.e., objects. So a composite component that contains component instances (often loosely called subcomponents) will have one member variable for each of its subcomponents. As a result there is a one-to-one mapping between the component instantiation hierarchy in the design and the object tree in the implementation.

Components can be instantiated more than once, in different parts of the design or within a single composite component. For the implementation of connectors (see below) it is necessary for those instances to have a unique (by component type) integer identifier (ID). This identifier (value starting at 0) are in fact assigned during code generation but, since they may depend on their parent ID, are set by their parent composite using their constructor.

The following example shows the code that is generated for a composite component and its subcomponents (CoCo input on the left; the generated code on the right):

```
component Composite {
    Sub s1;
    Sub s2;
    // connectors, etc.
}

component Sub {
    // ports etc.
}
```

```
class Composite_Base extends PecosPassiveComponent{
    private int id;
    private Sub s1, s2;
    protected CompositeBase(int _id) {
        id = _id;
        s1 = new Sub(0+_id);
        s2 = new Sub(1+_id);
    }
}

class Sub_Base extends PecosPassiveComponent {
    private int id;
    public SubBase(int _id) {
        id = _id;
    }
}
```

Remarks:

- For C++ member initializers should be used with the constructor, as in:
  `CompositeBase( int id ): id(_id), s1(0+_id), s2(1+_id) {}`
- Components that are instantiated only once throughout the design do not need an ID. However, since in general this cannot be known when designing a component, its constructor will always require a value to be passed.
- A possibly confusing but essential property of building the instance tree is worth noting: the component base classes do the instantiating, but the user component subclasses are the ones that are being instantiated. The final object tree thus consist of user component objects only.

☠ The subcomponent instances are **private** to make them inaccessible from the "user" component class.
☠ The base class should never be instantiated by itself. Therefore its constructor is made **protected**.

5.2 Ports and connectors

Ports in the Pecos model are the means to exchange data between components. They are tied together by connectors that allow a shared data implementation (see [Model]). So instead of ports representing a piece of data in a component that is copied along the connectors, they can be seen to read/write some common data location. In the implementation the data locations for all connections in the application will be collected in a single class: the *DataStore*. This DataStore will be looked at in more detail in section 5.4. For now it suffices to know that this DataStore contains arrays of these values.

For active and event components the story is a bit more complicated. Their `synchronize` method will run in their parent's thread while their `execute` in their own, each of which has its own data store. Another interpretation of this is that ports on active or event components have both an internal and an external value. So a connector on the inside will refer to a different data location than one on the outside. And methods will need to be provided to copy values back and forth between the two.

5.2.1 Passive component ports

Ports can be efficiently implemented by means of `put` and `get` methods that directly access the data in the DataStore. To connect two ports all that is required is to make sure that their methods refer to the same data location. Which, as the DataStore keeps the data in arrays, comes down to using the same index in all `put` and `get` methods of ports that are connected (directly or through hierarchy).

The direction of a port (input, output, or inout) determines the kind(s) of access method(s) that need to be generated (see the table in the next subsection).

The following example illustrates the principle of generating `get` and `put` methods:

```
component Composite {
    Sub1 s1;
    Sub2 s2;
    connector c(s1.in,s2.out);
}

component Sub1 {
    input  float in;
}

component Sub2 {
    output float out;
}
```

```
class Sub1_Base {
    ...
    protected float get_in() {
        return(DataStore.floats[DataStore.SUB1_IN]);
    }
}
class Sub2_Base {
    ...
    protected void put_out( float val ) {
        DataStore.floats[DataStore.SUB2_OUT]=val;
    }
}
class DataStore {
    static final int SUB1_in  = 0;
    static final int SUB2_out = 0;
    ...
    float[1] floats;
}
```

Remarks:

- The use of **static final** values as indices has two advantages. First the code for component classes (`Sub1` and `Sub2`) can be generated without having to wait until the connection allocation is done. Second it provides symbolic indices that can be used for debugging.
  - Any inefficiencies resulting from using symbols instead of constant values can be expected to be optimized away by the compiler.
  - In C++ the get and put methods can be **inline**d to avoid the function call overhead. Modern Java linkers would do the same.

If a component is instantiated more than once then for each instance the ports will be linked through different connectors. And then the simple, fixed indexing used above will not suffice anymore. This is where the instance ID comes in. Instead of directly indexing the DataStore a second level of indirection is introduced using the ID as an index. For instance if component `Sub2` would have more than one instance, the following implementation of `put_out` is generated:

```
DataStore.floats[DataStore.Sub2_out[id]] = val;
```

while `Sub2_out` would be defined as (assuming `Composite.s2` would be the first instance):

```
static final int[] Sub2_out = { 0, ... };
```

Of course this mechanism could be used to implement all ports. However, the additional indirection introduces an overhead that compilers would be hard pressed to circumvent. And since it is expected that only a few components will actually have multiple instances it is considered worthwhile having the code generator make a special case for them.

Access to ports needs to be restricted to the component itself. As the component class will be a subclass of the generated base class that has the get and put methods this can be enforced by making them **protected**.

5.2.2 Active and event component ports

Active and event components have ports with semantics that differ from the passive ones. Basically a single port refers to two data areas, an external one that is shared with its environment, and an private, internal one that, if composite, it shares with its content. In normal operation (e.g. from its `execute`) only the internal data can be accessed. It is only when the active component is allowed to synchronize that the external data is accessible.

The internal data can be accessed through the usual `get` and `put` port methods. There is no similar, direct access provided to the external data. Instead for each port `import` and/or `export` methods are provided that will copy port data between the two areas. These methods can only be used from the `synchronize` method of the active component.

So for a component to read the external value on a port it has to wait for the `synchronize` to `import` the value after which it can `get` the internal value. Conversely to write an external value it has to `put` it to the internal store and then wait for the `synchronize` to be able to `export` it.

Note that the `synchronize` method is provided by the designer. There are no rules about what it should contain. So there is no guarantee that external values will eventually synchronized with the internal ones, although of course it wouldn't make a lot of sense if it never would happen.
Some more will be said about this in section 7.3 below.

The following shows the code for the`import` and `export` methods for an inout port named `p` of active component `Comp` (instantiated only once):

```
protected void import_p {
    DataStore.val_int[DataStore.Comp_p] =
                DataStore.val_int[DataStore.CompX_p];
}

protected void export_p {
    DataStore.val_int[DataStore.CompX_p] =
                DataStore.val_int[DataStore.Comp_p];
}
```

The constant index `CompX_p` refers to the external value; `Comp_p`, like before, to the internal one.

Like for get and put methods the direction of the port determines what methods are provided. The following table give an overview of all port related methods.

Port direction	Passive Components	Active / Event Component
input	get_port	get_port

		`import_port`
`output`	`put_port`	`put_port` `export_port`

`inout`	`get_port`
	`put_port`
		`get_port`
		`put_port`
		`import_port`
		`export_port`

☠ The `import` and `export` methods may only be called from the `synchronize` method of the component. In C++ this can be enforced by making these methods **private** and `synchronize` a **friend**. Java seems to lack facilities to do something similar.

5.3 Data types

Besides providing basic type like integer, float, and byte, CoCo also allows the designer to define structured types. Furthermore, in the future, it will also allow making partial connections. The latter means that for instance a port that carries a structured type of two integers, can not just be connected to another port carrying the same type, but also, by using two connectors, to two separate single integer ports.

Base types can simply be implemented by creating a separate array for each of them in the DataStore. As is already shown in the example above the port just indexes the array that holds the proper value type.

An obvious mapping of structured types would be to create equivalent structures or classes in the target language and use, as for base types, separate arrays in the DataStore to hold their values. However such a solution would not be able to support the partial connections mentioned above. For instance, if an integer of one structured type would be connected to a plain integer, these values would reside in separate arrays and thus could not be shared. Some automated copying would need to be provided which is cumbersome and inefficient.

A better, but for a code generator much more elaborate, solution would be to split all structured types into their base constituents. That puts some burden on the generation of `get` and `put` methods as for those types they now have to collect the data from the DataStore and fill an appropriate structure.

🏳 As the `get` and `put` methods manipulate data by value (as opposed to by reference) they would require repeated copying of the struct. To avoid this unnecessary inefficiency it was decided to alter the get and put interface in C++ to having the user pass references to structured types to be read or filled by the port methods. In Java everything goes by reference already so here types have to be duplicated explicitly.

5.4 The DataStore

Most has already been said about the DataStore in the previous sections. In principle all it needs to contain are arrays to store port values in. Several arrays are needed if different value types are used on ports.

In addition, it is useful to have it contain index definitions, as it allows the generation of symbolic code which is useful for debugging. For this reason indices for connectors could also be added.
There is no essential need for these indices to be kept in the DataStore. They could also be stored in the component itself for instance. The reasons for choosing the former solution are related to the current implementation of the code generator and go beyond the scope of this document.

🏳 Although the data areas of active/event components need to be kept separate, there is no need to create distinct DataStore classes for each of them. In the implementation they can all be merged into a single class as long as the index generation algorithm guarantees that the indices between different data areas are distinct.

☠ Access to the DataStore should be prohibited except for the `put` and `get` methods. In Java this can be realized by giving the DataStore **package** visibility and keeping user defined classes in a separate package. In C++ the DataStore can be made **private**, specifying only the `get` and `put` methods as **friend**s.

6. Mapping the execution model

The Pecos model specifies three types of components: Passive, Active and Event components. Each component has two or three designated methods that will be used to invoke their functionality. For passive components they are `initialize` and `execute`; active and event components have an additional `synchronize`.
Running a Pecos application means invoking these pieces of functionality at the appropriate times.

Not surprisingly first the system needs to be initialized. This is performed by calling the `initialize` method of every component instance in the order described below. But the next step, running the system, is less straightforward. The `execute` and `synchronize` methods need to be run in all sorts of configurations: cyclic, asynchronously, in sequence, regularly spaced in time, etc.

The model [Model] allows no direct specification of what should run when. All that is provided are means to specify timing constraints, given as properties on components. It is currently being investigated whether a tool can be created that would "solve" these constraints and thus find an order of execution which can be expressed in RTE scheduling primitives. Such a tool could be considered part of language mapping, but is complex enough to be a project on its own. Therefore it will be documented separately and will not be discussed here.

However, CoCo has been extended with a representation of schedules. A schedule is a specification of what happens, i.e., in what order and when the execute and synchronize methods will be invoked within a single active component. This extension allows the constraint solving tool to be kept target language independent. Also, if such need arises, it allows the schedule to be tweaked or even completely provided by the user. The latter will, of course, be the only choice as long as no tool exists.
Allowing the designer this much control is not without danger: there is no guarantee that the schedule complies with the timing constraints. However, assuming the constraint solver can be made, this tool would be able to verify a schedule as well (add the schedule as an additional constraint). Furthermore the advantage of giving the designer flexibility and control prevailed in this tradeoff.

This section will discuss initialization and the mapping of the schedule syntax of CoCo to the runtime environment supporting running those schedules.

6.1 Initialization

The designer needs to specify a possible empty `initialize` method for each component. This method is meant only for internally initializing the component, but can, through a component's `put` methods also be used to initialize connector values.

Initialization takes place simply by a depth-first walk through the instance tree initializing the instances encountered in a post-order fashion. Meaning that an instance is only initialized after all its children have been. Children are initialized in the exact order in which they are declared in the component.
The process is achieved by providing the base components with a generated `_initialize` method that calls the same method on all its children followed by its own `initialize` method. The generated initialization code for the composite component in section 5.1 would look like this:

```
class CompositeBase extends PecosPassiveComponent{
    ...
    void _initialize() {
        s1._initialize();
        s2._initialize();
        initialize();
    }
}
```

Note that this schema makes no difference between active and passive components. So it is assumed to take place before the different threads that will run the application are fired up.

Having this code in place all that is needed to initialize the system is to call the `_initialize` method of the top-level component.

☠ Since initialization is performed by the system at startup it would be desirable to make `_initialize` less public. In C++ this can be achieved by making the methods **private** and a component a **friend** of all its subcomponents.
In Java the `_initialize` routines can easily be protected by giving them **package** scope. Assuming all generated code is contained in a separate package this will automatically do the right thing.

6.2 Implementing schedules

After initialization the application is ready to run. The RTE provides three simple primitives (implemented in classes) to control the execution of component behavior: Task, Job, and Activity. See [RTE] for a more detailed description.
Schedules specified in CoCo can be mapped almost one-to-one to the RTE. The only difference is that the syntax allows the specification of "anonymous" jobs, for which in the code explicit jobs need to be created. An example is `sched$0` in the following:

```
schedule sched of top
         every 100 at 1 {

    job1 = {
        exec comp1;
        sync comp2;
    };

    job1 at 0;
    exec comp3 at 50;
}
```

```
static final Activity[] sched$job1 = {
    new Activity( comp1, EXEC ),
    new Activity( comp2, SYNC )
};
static final Activity[] sched$0 = {
    new Activity( comp3, EXEC ),
};
static final Job[] sched$$ = {
    new Job( sched$job1, 0 ),
    new Job( sched$0, 50 )
};
static Task sched = new Task( sched$$, 100, 1 );
```

A running field device does not dynamically alter the way it runs. Therefore the schedules can be implemented using **static final** tables. As shown in section 7.1 schedules are kept in the PecosDevice class.

6.3 Event components

Event component have been more or less ignored in the discussion above. The reason is that in general they will be small non-composite components that need to quickly react to an event. In that case all that needs to be done is to install the `execute` method as the event handler.

Of course, this need not always be true. In case an event component is composite a schedule needs to be provided. Then the schedule itself could be executed upon receipt of the event. The only difference with the above would be that the schedule would not be periodic.

7. Miscellaneous

7.1 The PecosDevice

Given all the code that is generated above and the code that is provided by the user, there is still one piece missing: a central control unit. This is provided in the form of a separate class called PecosDevice.
This class provides the following:

- The instantiation of the top level component
- The implementations of the schedules
- A method `initialize` that initializes the top-level component and all schedules.
- A method `run` that will bring the system to life by starting the threads for each schedule.

So a PecosDevice looks like this (assuming top level component is named `TopComponent`):

```
class PecosDevice {
    TopComponent top = new TopComponent(0);
    ...schedule declarations as above...
    public void initialize() {
       top._initialize();
       sched1.create();
       ...
    }
    public void run() {
       sched1.start();
       ...
    }
}
```

Everything in PecosDevice can be implemented as static (class) variables so there is no need to instantiate the class. Having this class one minimally needs to do to get the device running is to provide the following main():

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

This last step is again left to the designer to allow more flexiblity. For instance, it might be necessary to perform initializations on user provided support classes.

7.2 Properties

There are two ways of looking at properties. There is the model view where a couple of well-defined properties are actually part of the model. On the other hand, and this is the view taken here, they can be seen as additional information attached to the model to be used by tools that run on the model. One such tool would be a verification tool, another the code generator. In this view it is basically up to the tool to decide what properties it is interested in and possibly define its own properties.

The predefined property bundles, see [Model], have no direct application for the code generator, and are thus ignored. Currently the code generator does not honor other properties.

7.3 User code

User code refers to the code, written in the implementation language, that needs to be added to the generated code to get a working system. It comprises the local functionality of components and the implementation of synchronization protocols between threads. These are specified by defining the `execute` and `synchronize` methods of the component classes. Of course, if necessary, support classes, data structures, and functions can be added to this.

As shown in the class diagram in Figure 1 each generated component base class needs to be subclassed by the user. If the component has no behavior itself then such a subclass would look like this
(active components would also have a `synchonize`):

```
class Component extends ComponentBase {
    public void Component( int id ) {
        super( id );
    }
    public void initialize() {}
    public void execute() {}
}
```

This would be a valid implementation for a purely composite, passive component. Leaf components would at least have some function in the `execute` method.

Active components, assuming they communicate with their environment,always need to specify function. If nothing else, the synchronization with the component's environment by means of importing or exporting port values (see 5.2.2) needs to be given. A nice feature of the model is that synchronization always takes place within a single (active) component, that is, between the `synchronize` and `execute` methods . Synchronization including things like resource locking takes only place between the `synchronize` and `execute` methods of an active component and possibly its subcomponents.

Note that entering code into the `execute` and `synchronize` methods does not mean that they will be invoked at some point in time. For that they need to be mentioned explicitly in the schedule (see section 6.2). For `initialize` nothing needs to be done.

8. Special purpose code generation

This is future work and will be filled in later.

9. Summary

This deliverable describes how an application modelled using the Pecos component model can be mapped to an C++ or Java execution environment. It shows how, with ease, modeling entities are mapped onto target language constructs, how they can be combined with user specified functionality, and how the whole can be made to run. Moreover, it shows that the implementation can be efficient and, with use of some standard OO protection techniques, quite robust.

In short, it shows that, in practice, Pecos applications can be turned into executables.

10. References

[C++] Bjarne Stroustrup, "The C++ programming Language (3rd edition), Addison Wesley Longman, Reading, MA, 1997.

[CoCo] Genzler, Winter, Christof, "Description of the CoCo architectural description language and composition environment", IST-1999-20398 Pecos Deliverable D2.2.5, May 2002

[Java] James Gosling, et. al., "The Java Language Specification", second edition, http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html Sun Microsystems, 2000

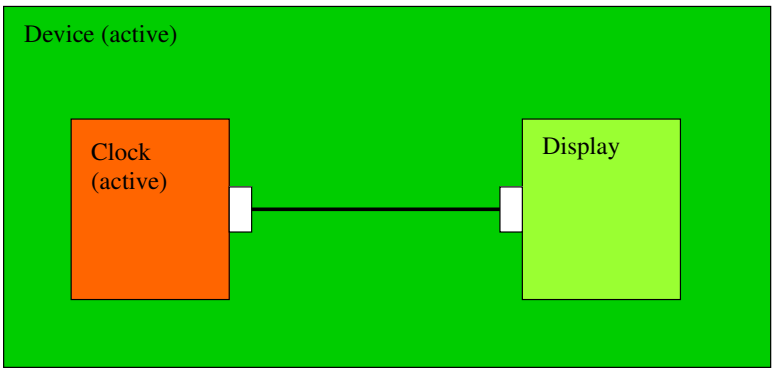
[Model] Roel Wuyts, et. al., "Field-Device Component Model-V", IST-1999-20398 Pecos Deliverable D2.2.8-5, Oct 2001

[RTE] Bastiaan Schönhage, "Pecos Run-time Environment, C++ and Java", IST-1999-20398 Pecos Deliverable D4.6-1, May 2002

Appendix A. C++ code generation example

Below a very simple device will be specified and code resulting from running the current implementation of the code generator will be shown. It is just intended to give a taste of what the tool does. No attempt is made to show all aspects that were discussed above.

The device consists of three components: a main composite component with two subcomponents. Of the two subcomponents one is active and reads the system time. The other is passive and prints the time.
The device looks like this



The following are the component specifications provided by the designer:

The Device component:

CoCo

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

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

C++

```
#include <DeviceBase.h>

class Device: public DeviceBase {
    public Device( int id ): DeviceBase(id) {}

    public void initialize() {}
    public void execute() {}
    public void synchronize() {}
}
```

The Clock component:

CoCo

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

C++

```
#include <ClockBase.h>

class Clock: public ClockBase {
    public Clock( int id ): ClockBase(id) {}

    public void initialize() {
        put_sec( 0 );
    }
    public void execute() {
        put_sec( time());
    }
    public void synchronize() {
        export_sec();
    }
}
```

The Display component:

CoCo

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

C++

```
#include <DisplayBase.h>

class Display: public DisplayBase {
    public Display( int id ): DisplayBase(id) {}

    public void initialize() {}
    public void execute() {
        long secs;
        get_time_in_secs( &secs)
        printf( "It is now: %s\n", localtime( secs ));
    }
}
```

The Schedules:

CoCo

```
schedule sched1 of Device every 1000 at 10 {
   {
    sync display;
    exec clock;
   } at 10;
}
schedule sched2 of Clock every 1000 at 1 {
   {
    exec;   // executes Clock
   } at 0;
}
```

The following is the code that has been generated (reformatted to fit the page):

DeviceBase.h

```
#ifndef _DEVICEBASE_H
#define _DEVICEBASE_H

#include <component/PecosComponent.h>
#include "DataStore.h"
#include "Clock.h"
#include "Display.h"

class DeviceBase: public PecosActiveComponent {

  public: DeviceBase( int id ): PecosActiveComponent("Device")
      ,clock(0)
      ,display(0)
  {}

    public: Clock clock;
    public: Display display;
    public: void _initialize() {
      clock._initialize();
      display._initialize();
      initialize();
    }
}; /* DeviceBase */

#endif /* _DEVICE_H */
```

ClockBase.h

```
#ifndef _CLOCKBASE_H
#define _CLOCKBASE_H

#include <component/PecosComponent.h>
#include "DataStore.h"

class ClockBase: public PecosActiveComponent {

  public: ClockBase( int id ): PecosActiveComponent("Clock")
  {}

  protected: inline void put_secs(long val) {
     DataStore::val_long[DataStore::Clock$secs] = val;
  }
  protected: inline void export_secs() {
     DataStore::val_long[DataStore::Clock$X_secs]
=DataStore::val_long[DataStore::Clock$secs];
  }
    public: void _initialize() {
      initialize();
    }
}; /* ClockBase */

#endif /* _CLOCK_H */
```

DisplayBase.h

```
#ifndef _DISPLAYBASE_H
#define _DISPLAYBASE_H

#include <component/PecosComponent.h>
#include "DataStore.h"

class DisplayBase: public PecosPassiveComponent {

  public: DisplayBase( int id ): PecosPassiveComponent("Display")
  {}

  protected: inline void get_time_in_secs( long *val) {
     *val = DataStore::val_long[DataStore::Display$time_in_secs];
  }
    public: void _initialize() {
```

	`initialize();`
	`}`
	`}; /* DisplayBase */`
	
	`#endif /* _DISPLAY_H */`

DataStore.h	`#ifndef _DATASTORE_H`
	`#define _DATASTORE_H`
	
	`class DataStore {`
	`public: static const int Display$time_in_secs=0;`
	`public: static const int Clock$X_secs=0;`
	`public: static const int Device$time=0;`
	`public: static const int Clock$secs=1;`
	`public: static long val_long[2];`
	`}; /* DataStore */`
	
	`#endif /* _DATASTORE_H */`

DataStore.cpp	`#include "DataStore.h"` `long DataStore::val_long[2];`

PecosDevice.h	`#ifndef _PECOSDEVICE_H`
	`#define _PECOSDEVICE_H`
	
	`#include "Device.h"`
	`#include "scheduler/Task.h"`
	`class PecosDevice {`
	`private: static Device top;`
	`private: static const Activity sched2$0[];`
	`private: static const Job sched2$schedule[];`
	`private: static Task sched2;`
	`private: static const Activity sched1$0[];`
	`private: static const Job sched1$schedule[];`
	`private: static Task sched1;`
	`public: static void initialize() {`
	`top._initialize();`
	`sched2.create("sched2", 1000);`
	`sched1.create("sched1", 1000);`
	`}`
	`public: static void start() {`
	`sched2.start();`
	`sched1.start();`
	`}`
	`}; /* PecosDevice */`
	
	`#endif /* _PECOSDEVICE_H */`

PecosDevice.cpp	`#include "PecosDevice.h"`
	`Device PecosDevice::top(0);`
	`const Activity PecosDevice::sched2$0[] =`
	`{Activity(&PecosDevice::top, EXEC),`
	`Activity(NULL)};`
	`const Job PecosDevice::sched2$schedule[] =`
	`{Job(sched2$0, 0),`
	`Job(NULL)};`
	`Task PecosDevice::sched2(sched2$schedule, 1000, 1);`
	
	`const Activity PecosDevice::sched1$0[] =`
	`{Activity(&PecosDevice::top.clock, SYNC),`
	`Activity(&PecosDevice::top.display, EXEC),`
	`Activity(NULL)};`
	`const Job PecosDevice::sched1$schedule[] =`
	`{Job(sched1$0, 10),`
	`Job(NULL)};`
	`Task PecosDevice::sched1(sched1$schedule, 1000, 10);`