Progress Component Model Reference Manual

version 0.5

Tomáš Bureš, Jan Carlson, Ivica Crnković,
Séverine Sentilles and Aneta Vulgarakis

April 4, 2008

Abstract

This report describes the component model developed within Progress. In addition to defining the syntax and semantics, it also gives some background and motivation, and describes how this work relates to the overall Progress vision and to the work in other areas of the project.

Contents

1 Introduction **3**
1.1 Conceptual development framework 3
1.2 Component model overview 4

2 System design level **5**
2.1 Subsystem component 5
2.2 Connecting subsystems 6
2.3 Primitive and composite subsystems 6

3 Subsystem design level — ProSave **7**
3.1 Components 8
3.1.1 Services, groups and ports 8
3.1.2 Component semantics 9
3.2 Connecting components 9
3.2.1 Connections 10
3.2.2 Connectors 10
3.3 Primitive and composite components 12
3.3.1 Primitive components 12
3.3.2 Composite components 13
3.4 Using ProSave components in a subsystem 14
3.5 Abstract execution semantics 15

4 Meta-model **17**
4.1 System level 17
4.2 ProSave level 19

5 Example **26**

1 Introduction

The component model described in this report is developed within PROGRESS[1], a strategic research centre funded by the Swedish Foundation for Strategic Research. The key objective of PROGRESS is to apply a software-component approach to engineering and re-engineering of embedded software systems, in particular within the vehicular, telecom, and automation domains.

The component model is influenced by previous work in the SAVE project[2] and also to some extent by the Rubus component technology [3]. However, compared to these projects, it provides a stronger concept of compositional and reusable components.

1.1 Conceptual development framework

The PROGRESS concept paper [2] identifies three major activities in the component-based development process for embedded systems. These activities are shown in Figure 1. (Note that this picture together with the anticipated levels will most probably change in the next version of the concept paper.)

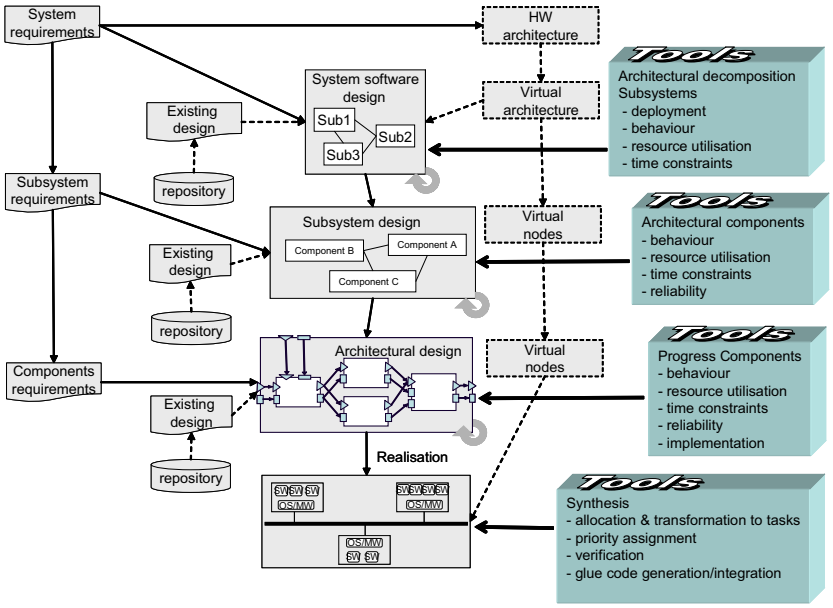


Figure 1: Conceptual design framework.

At the top level (*system software design*), the system is split into subsystems (e.g., ABS, engine control, etc.), which are represented by coarse-grained compo-

[1] PROGRESS homepage: `http://www.mrtc.mdh.se/progress`
[2] SAVE homepage: `http://www.mrtc.mdh.se/save`

nents with fairly limited interaction capabilities (typically to just asynchronous transmission of data).

In *subsystem design* the internal design of each individual subsystem is elaborated. At this level, the partitioning of functionality into smaller units is done at a high level of abstraction, and might not correspond to concrete, distinguishable parts in the final system. We envision that this level will be used mainly when designing complex sub-systems. In case of small sub-system, this step may be skipped – meaning that the internals of a subsystem would be directly modeled on the architectural design level.

The third step, *architectural design* defines the concrete realisation of the subsystem by means of interconnected software components. The aim of this level is to have very concrete and low-level model of a subsystem so that various analyses may be performed and also that *realization* (the fourth step) is achievable.

At all three levels, the component modeling is connected with the use of a repository and with various analysis techniques. Components may be developed from scratch or they can be retrieved from a repository and reused. This allows for two types of development: top-down and bottom-up. In fact, we anticipate that the real component development will be a mixture of these two. That means that existing components will be reused as they are or with slight modifications, and by composing them the complex functionality will be derived (i.e., the bottom-up approach). Components which could not be found in the repository will be developed from scratch in top-down manner. This is further elaborated in the concept paper [2].

In parallel to such functional decomposition, there are also activities related to *deployment*. They focus on modeling the target platform and on mapping components (i.e., design-level functional units) to a final system consisting of units of distribution and execution.

The whole development is accompanied by analysis, which allows estimating execution times and memory footprint, reliability attributes, etc.; thus guiding the design by justifying particular design decisions or providing early warnings.

1.2 Component model overview

This report presents a two-layer component model. The top layer corresponds to the system design level in the conceptual framework, and here the system is modelled as a number of active and concurrent subsystems, communicating by message passing. The second layer, called ProSave, addresses the internal design of a subsystem down to primitive functional components implemented by code. In the conceptual framework, this corresponds to the layers of subsystem design and architectural design, but ProSave does not distinguish between the two. Contrasting subsystem components, ProSave components are passive and the communication between them is based on pipes-and-filters.

In both layers, information about a component is stored along with the components in the repository, including requirements, textual documentation and models of the behaviour and resource usage. Since it is anticipated that

additional analysis techniques will be developed in the future, the repository structure is extendable, so that additional information required by a new analysis method can be added without impacting existing components.

The system layer and ProSave are described in Section 2 and Section 3, respectively. Section 4 presents the meta-model, formalising the concepts of the component model and how they relate. Section 5 provides a larger example of a subsystem designed in ProSave.

2 System design level

At the system level a component represents a whole subsystem. This means that it is fairly independent and has its own activities. The communication between components is typically asynchronous and realized by message exchange. Thus for the system level we use a relatively simple component model with asynchronous message passing as the communication paradigm. The component model is hierarchical, meaning that a subsystem can internally be realised by a collection of communicating smaller subsystems [3]

The components on this level are often meant to be allocated to different nodes in a distributed system. Even a single subsystem may consists of parts that end up on different nodes. The distribution is however not specified in this component model, as it is provided by a separate deployment model.

2.1 Subsystem component

A subsystem is represented by a component with typed input and output message ports, as shown in Figure 2. The ports express what messages the subsystem recieves and sends. The external view also includes attributes and models.

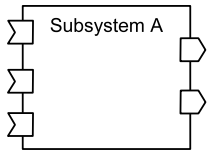


Figure 2: External view of a subsystem with three input message ports and two output message ports.

Subsystems are active, in the sense that they may include activities that are performed periodically or in response to some internal event, rather than as the result of an external activation. They can contain reactive parts as well, that are performed in response to the arrival of a message.

[3] The details of this hierarchy has not been elaborated yet.

2.2 Connecting subsystems

A system consists of a collection of subsystems and connections from output to input message ports. Message ports are not connected directly, but via a *message channel*. This means that it is possible to define that a particular data, e.g., speed, will be required in the system before the producer and recievers of this data are defined. Also, it is possible to specify that two subsystems use the same data as input, before the producer of the data has been defined. Figure 3 shows an example. Message channels support n-to-n communication, i.e., several output ports as well as several input ports can be connected to the same channel.

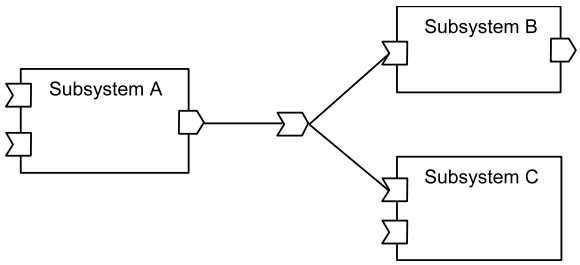


Figure 3: Three subsystems communicating via a message channel.

2.3 Primitive and composite subsystems

Primitive subsystem can be internally modelled by ProSave components, as described in Section 3.4. Alternatively, they can be realised by code conforming to the runtime interface of PROGRESS subsystems[4]. In the case of legacy code, i.e., existing code developed outside the PROGRESS context, some modifications or additions would typically be required to make it compatible with the PROGRESS subsystem interface. This *componentisation* activity is described in the concept paper [2].

A composite subsystem internally consists of subsystems and local message channels. There are also connections that associate local message channels with message ports of the composite subsystem or the subsystems inside. This allows an input port, acting as a message consumer outside the component, to act as a message producer internally. Oppositely, an output port consumes messages on the inside and act as a message producer from the outside.

Two message channels connected to the same message port, outside and within the component, respectively, will typically not manifest as two separate entities in the final system. Rather, this connection via the message port can be

[4]The details of this runtime interface remains to be decided, as a part of the work on deployment and synthesis.

seen as a way to "unify" a channel defined locally within a composite component with a particular channel in the component environment[5].

An example of a composite subsystem is given in Figure 4.

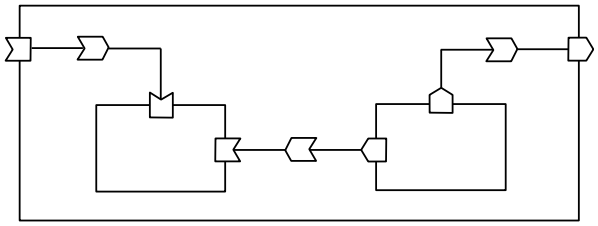


Figure 4: An example of a composite subsystem.

3 Subsystem design level — ProSave

Internally, subsystems can be designed in different ways, as long as they conform to the subsystem semantics, e.g., that inter-subsystem communication is based on message passing as described in the previous section. This section defines *ProSave*, a component-based design language especially targeting subsystems with complex control functionality. It defines the ProSave constructs and their semantics, and describes the connection between ProSave and the system level concepts.

In ProSave, a subsystem is constructed by hierarchically structured, interconnected *components*. Components are *passive*, meaning that they do not contain their own execution threads and thus can not initiate activities on their own. Instead, they remain passive until activated by some external entity, and when activated they perform the associated functionality before returning to the passive state again. Component activation is always initiated at the subsystem level, where components can be associated with periodical activation or the occurrences of some external or internal event. This is further discussed in Section 3.4.

ProSave components are design-time entities that are typically not distinguishable as individual units in the final executing system. During the deployment and realisation process, the components are transformed into executable units, e.g., tasks, in order to achieve the desired runtime efficiency by avoiding a costly component framework at runtime.

The component model is based on a pipes-and-filters architectural style, but there is an explicit separation between data transfer and control flow. The

[5]Currently, the component model does not contain any constructs that modify messages as they pass the boundary of the enclosing subsystem, for example queuing incoming messages that are to be delivered to an internal subsystem that does not support queuing. This type of construct might be introduced later on, if it is required for the development scenarios we envision.

former is captured by *data ports* where data of a given type can be written or read, and the latter by *trigger ports* that control the activation of components.

3.1 Components

A ProSave component is a reusable unit of functionality. Its task is to encapsulate relatively small and rather low-level functionality, thus it may not be distributed.

The external view of a component consists of two major parts: *ports* through which the functionality provided by component can be accessed, and information about the component, represented by structured *attributes*.

Internally, the functionality of a component can either be realised by code, or by interconnected sub-components, but the distinction is not visible from the outside. This black-box view of a component, based only on the externaly visible structure and attributes, is useful since it facilitates compositional reasoning and supports early analysis of systems when some components are yet to be implemented. Still, some analysis might require or benefit from a more detailed information than what is provided by the external view. In this case, nothing prevents it from examining the contents of a component, e.g., the sub-component structure. In particular, synthesis activities assume a fully defined system, and will probably mostly adopt a white-box view to allow optimisations spanning several levels of nesting.

3.1.1 Services, groups and ports

The functionality of a component is made available to external entities by a set of *services*, each corresponding to a particular type of functionality that the component provides. Each service consists of the following parts:

- An *input port group* consisting of a *trigger port* by which the service can be activated and a set of *data ports* corresponding to the data required to perform the service.
- A set of *output port groups* where the data produced by the service will be available. Each group consists of a number of *data ports* and a single *trigger port* indicating when new data is available.

Each port belongs to a single group, and each group belongs to one service. The ports of an input group are informally refered to as input ports, and ports of output groups are called output ports. Figure 5 illustrates these concepts.

Data ports are typed and associated with a default value used for initialisation. The type is specified by a type definition in C.[6]

In addition to the input trigger ports and the related entities, a component has a collection of *attributes*. Some of them are explicitly associated with a specific port, group or service (e.g., the worst case execution time of a service,

[6] Elaborating on the details of a suitable type system is included in the plan for future improvements.

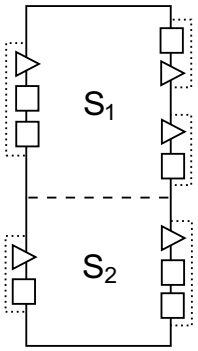


Figure 5: External view of a component with two services; S_1 has two output groups and S_2 has a single output group. Triangles and boxes denote trigger ports and data ports, respectively.

or the range of values produced at a data port), while others are related to the component as a whole, for example a specification of the total memory footprint.

3.1.2 Component semantics

Initially, all services of a component are in an inactive state where they can recieve data and trigger signals to the input ports, but no internal activities are performed. When an input trigger port is activated, all the data ports in the group are read in one atomic operation and then the service switches into an active state where it performs internal computations and produces output at its output groups. The data and triggering of an output group are always produced in a single atomic step. Before the service may return to the inactive state again, each of the associated output groups must have been activated exactly once.[7]

It is assumed that a service is not triggered again while in the running state. To avoid inefficency, we envision that this is ensured by analysis at design time, rather than by a runtime mechanism, and thus the result of triggering a running service is not specified by the component model.[8]

3.2 Connecting components

Components can be connected to collaborate in providing more complex functionality. This is done by simple connections that transfer data or control and by additional connectors providing more elaborate manipulation of the data-

[7]The requirement that all output groups must be activated might be relaxed in the future, if optional output groups are introduced.

[8]If this is determined to be too weak, future versions of ProSave might specify that triggering reaching an active service should be ignored.

and control flow. Connected components can be found inside composite components, and on the top level inside subsystems.

3.2.1 Connections

A *connection* is a directed edge which connects two ports — either input data port to output data port or input trigger port to output trigger port. In the case of data ports, they must have compatible types. Graphically, a connection is represented by an arrow from output- to input port.

There can be at most one connection attached to a port. An exception to this rule is that the ports of a composite component can have one connection on the inside and one on the outside.[9]

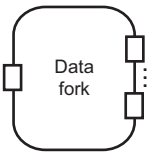
A connection between data ports denotes the data transfer. ProSave follows the push-model for data transfers. It means that whenever data is produced on a data output port, the data is transfered by the connection to the input data port and stored there. The triggered component or connector always uses the latest value written to the input data port.

A connection between trigger ports transfers the control flow. That means that a trigger port on the target endpoint of the connection is triggered as the result of the trigger port on the source endpoint of the connection being activated.

In general, a transfer is not an atomic action, and the transfer over two different connections can be carried out concurrently or in arbitrary order. However, there is one exception to this, described in more detail in Section 3.5. This exception essentially specifies that when data and triggering appear together at an output port group, the data should always be delivered before the trigger transfer starts.

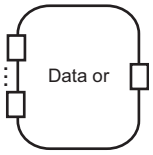
3.2.2 Connectors

In addition to connections, there are constructs called *connectors* that may be used to control the data- and control-flow. In general, a connector is represented by a rounded rectangle with the name of the connector written inside. The most used connectors may also have a simplified notation. This is the case of Data fork and Control fork, which may be abbreviated using a thick dot.

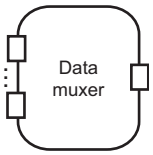


A *data fork* is used to split a data connection to several ones. It has one input data port and at least two data output ports. Data written to the input port are duplicated on the output ports. Graphically, this connector can also be denoted by a thick dot.

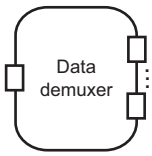
[9] Strictly speaking, we should say one connection on the inside of the component and one on the outside of each *instance* of the component. This is further discussed in Section .



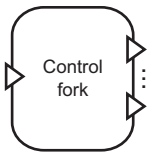
The *data or* connector is used to merge several data connections to one. It has one output data port and least two data input ports. Data written to any of the input ports are forwarded to the output port.



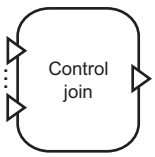
A *data muxer* allows grouping several data inputs into one output. It is mainly used to build data of a message (exchanged at a system level). It has two or more input data ports and one output data port. The type of the output data port is a struct comprising data of all input data ports. Whenever data is written to an input data port, it updates it in the relevant parts of the output data stuct and makes the data visible on the output data port.



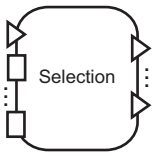
A *data demuxer* works as an inverse to data muxer. It has two or more output data ports and one input data port. Whenever data is written to the input data port, it is extracted and respective parts made available on the output data ports.



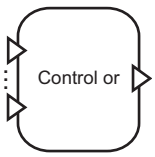
A *control fork* is used to split control flow to several concurrent paths. This connector has one input trigger port and at least two output trigger ports. Whenever the input trigger port is triggered, the trigger is transfered to all output trigger ports. Graphically, this connector can also be denoted by a thick dot.



The *control join* connector joins the control flows of several concurrent paths (an inverse operation to Control fork). This connector has one output trigger and at least two input trigger ports. It waits until all input trigger ports are triggered, then it triggers the output port.



Selection is used to choose a path of the control flow depending on a condition. This connector has one input trigger port, and several output trigger ports and at least one input data port. The connector has associated conditions over the data coming from the input data ports. Based on the result of evaluating the conditions, it forwards the incoming trigger to exactly one of the output trigger ports.[10]



The *control or* connector is used to join control flows of alternative paths (an inverse operation to Selection). The connector has at least two input trigger ports and one output trigger port. It forwards each incoming trigger to the output trigger. In contrast to control join, it does not wait for all input triggers to become triggered.

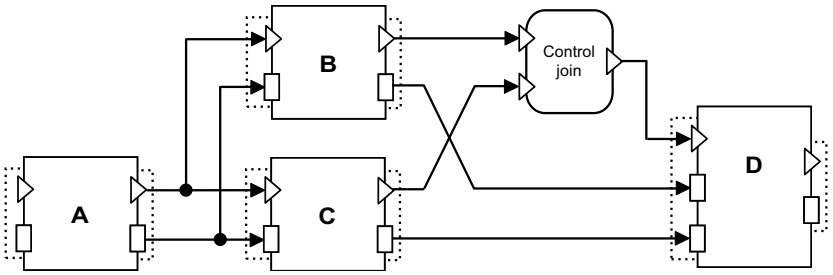


Figure 6: A typical usage of fork- and join connectors. When component A is finished, components B and C are executed in arbitrary order (possibly interleaved). Component D is executed once both B and C have finished.

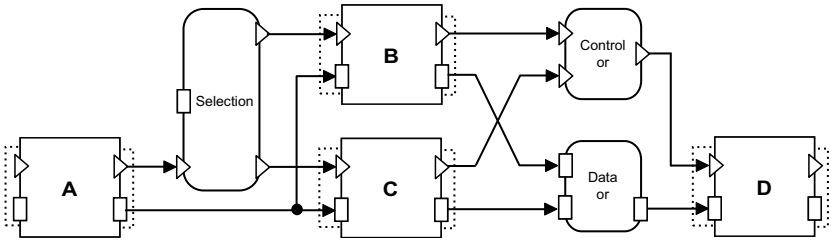


Figure 7: A typical usage of selection and or-connectors. When component A is finished, either B or C is executed, depending on the value at the selection data port. In either case, component D is executed afterwards, with the data produced by B or C as input.

The list of connectors is presumably incomplete and may grow over time as additional data-/control-flow constructs prove to be needed. Figures 6 and 7 show two typical usages of connectors.

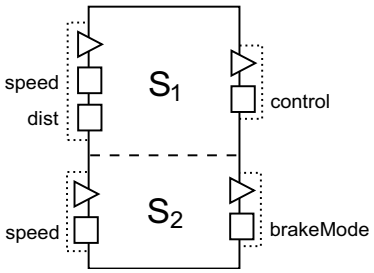
3.3 Primitive and composite components

When considering the internal structure, components come in two types: *primitive* components which are realised by code, and *composite* components which consist of internal components that together provide the desired functionality.

3.3.1 Primitive components

Primitive components may consist of several services, but they are restricted to have at most one output port group for each service.[11] The behaviour of each service is realised by a non-suspending C function. In addition, the component

[11] If primitive components with multiple services turn out to be problematic for synthesis, they might be restricted to have only a single service.




```
typedef struct {
  int *speed;
  float *dist;
} in_S1;

typedef struct {
  int *control;
} out_S1;

typedef struct {
  int *speed;
} in_S2;

typedef struct {
  int *brakeMode;
} out_S2;

void init();
void entry_S1(in_S1 *in, out_S1 *out);
void entry_S2(in_S2 *in, out_S2 *out);
```

Figure 8: Example of header file for a primitive component with two services, and no explicit name mappings.

has an init function which is called at system startup to initialise the internal state.

More concretely, the primitive component specifies a header C file and a source C file, where the init function and the service entry functions are declared and defined. The header file also declares the structs used for input and output to the services. By default, the naming of entry functions and argument structs is based on the names of services and ports, but explicit name mappings can also be supplied (see 4.2). Figure 8 shows an example of a header file.

3.3.2 Composite components

The internal view of a composite component consists of sub-component instances, connections and connectors. Each sub-component instance (or sub-component for short) is realised by a primitive or composite component, developed either from scratch or retrieved from repository. Connections and connectors control the order in which sub-components are invoked and how data are exchanged among them.

The ports of the encapsulating composite component appear "inwards" with the opposite direction — for example an input trigger port of the enclosing component acts as an output trigger port when seen from inside. That allows us to define the connections as always going from an output port to an input port.

When the component writes to an output port, this data does not become available outside the component until the trigger port of the port group is activates. When this happens, the values of all data ports in the group atomically

appear on the outside. Similarly, the input data ports can receive data also when the service is in the active state, but these data are not propagated inside the component until the next time the service is activated.

The usage of sub-components, connections and connectors actually form workflows starting in an input trigger port of the composite component and ending in the output trigger ports. If the component has several services, then each service has its own workflow. It should not happen that a workflow triggers an output trigger of another service. To prevent such problems, ProSave limits internal interactions between services to only data connections (i.e. no triggering).

There are no additional restrictions imposed by the component model on the internal architecture of a composite component. Obviously, an incorrect use of the connections and connectors may produce an architecture which exhibits behavior forbidden for a component. Basically, we leave this as the responsibility of the component developer. However, we envision tool support and analysis methods that would allow a developer to validate an architecture and discover such faulty behavior.

3.4 Using ProSave components in a subsystem

ProSave serves for low-level modeling of a subsystem. Components in ProSave are passive, typically local and they communicate via data exchange and triggering. A subsystem differs from a ProSave component in several aspects. A subsystem has its own threads of execution, which means that it can actively initiate the execution of a particular functionality. Moreover, subsystems use message passing with explicit message channels as the means of communication, and parts of a single subsystem often end up on different nodes in the final system.

This section describes how ProSave can be used to define the internals of a subsystem. This is done in a similar way to how composite components are defined internally — as a collection of interconnected components and connectors — but with some additional connector types. These connectors allow for:

a) mapping between message passing (used at the system level) and trigger/data communication (used in ProSave), and

b) specifying activation of ProSave components, either periodically or as the result of an external event.

The additional connectors are described in detail in the list below.

Externally, an *input message port* acts as a subsystem input message port. Internally, it has one output trigger port and one output data port which can be connected to a ProSave component or connector. Whenever a message is received, the message port writes message data to the output data port and activates the output trigger.

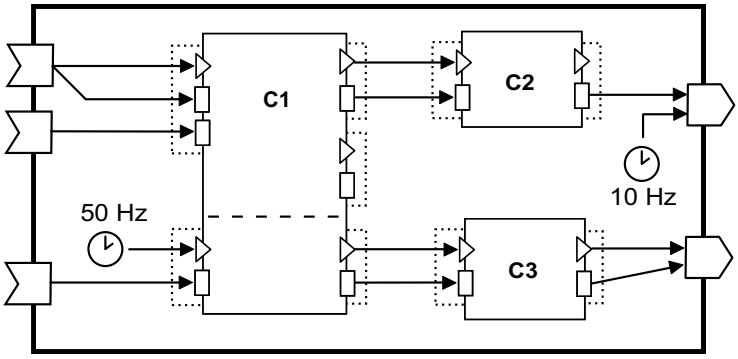
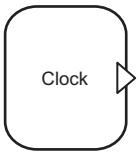


Figure 9: A subsystem internally modelled by ProSave.

The *output message port* is an inverse to the input message port. It acts externally as a sub-system output message port. Internally, it exhibits one input trigger and one input data port. When the trigger is activated, the port sends a message with the data currently available on the input data port.



A *clock* serves for generating periodic triggers. It has one output trigger port which is triggered at a specified rate. Clocks are assumed to follow a common conceptual time, i.e., they are not allowed to drift. However, it is not assumed that all clocks produce their first activation simultaneously, meaning that the relative phasing between clocks is arbitrary. As an alternative notation, this connector can be represented by a clock symbol.

The coupling between ProSave and the system level is performed only at the top top level in ProSave, which means that the connectors listed above are not allowed inside a ProSave component.

The use of these connectors is exemplified in Figure 9. The encapsulating subsystem has message ports as described in Section 2. Internally, each message port acts as a trigger and a data port, which can be connected to other components or connectors in the ProSave way. Additionally, it is possible to use clocks for generating periodic triggers.

3.5 Abstract execution semantics

Parts of the semantics has been presented in previous sections, including the behaviour of a component as viewed from the outside and the meaning of the different connectors. This section gives a more complete view of the ProSave execution semantics.

The execution semantics follows the component hierarchy, meaning that it is defined for a single level of nesting, either inside a subsystem or inside a composite component. This allows reasoning about the behaviour of a system also when some components are not fully decomposed down to primitive components.

Note that the semantics defines activities and communication on a conceptual level, and is not meant to illustrate the concrete runtime communication mechanisms. During synthesis, the design-time components are transformed into runtime entities, such as tasks, with different communication possibilities. It is the responsibility of synthesis to ensure that the behaviour of the runtime system is consistent with what is specified by the execution semantics and the ProSave design. For example, although the semantics view data transfer on different levels of nesting as separate activities, the final system may realise communication between two primitive components on different levels by a single write to a shared variable, ignoring the intermediate steps of activating input and output port groups, as long as the overall behaviour is consistent with the execution semantics.

For simplicity, we consider first the case of a composite component, and later extend this to the subsystem case. The overall responsibility of a composite component is to realise the internal workflows defined by connections, connectors and subcomponents. Concretely, this amounts to transfering data and triggering over connections, carrying out connector functionality and interacting with constructs one level of nesting above and below.

Seen from inside, data and triggering appear at the ports of the input port group when a service is activated. When this happens, or when new data or triggering become available at the output port of a subcomponent or connector, it should be forwarded on the connection leading out of the port. This transfer may take an arbitrary amount of time, and different transfers may be performed concurrently or in any order. There is only one restriction, related to the end-to-end delivery of the data and triggering of a single activation of an output port group: *The transfer of the trigger signal should not start before all data has arrived to its end destinations (i.e., to component ports).* Informally, this should hold also if the data passes through a connector that modifies it, such as a data demuxer.

The final phase of a transfer depends on the destination:

- When data reaches a port, it overwrites the current value of that port. In the case of a connector, the data is handled according to the connector semantics (e.g., written to the connector ouptut ports in case of a *data split*), otherwise nothing more happens.

- When triggering reaches a connector, it is handled according to the connector semantics.

- When triggering reaches a component input port, nothing happens if the service is currently active. If it is currently passive, then the values of the data ports of the triggered port group are atomically copied inside the component, and the service becomes active.

- When triggering reaches an output port of the enclosing component, the current contents of the ports of that group become available at the next level of nesting, possibly after some delay.

It is also the responsibility of the composite component to turn a service back to the passive mode once all the activities related to the activation of the service have finished. This means that there should be no pending transfer of data or triggering, and all subcomponent services that was activated should have returned to a passive state.

The semantics of the top level inside a subsystem is more or less the same as that of composite components. A transfer activity is initiated when data or triggering appears at an output port of a componentor connector, or at an input message port. When triggering reaches an output message port, the current data of that port is sent as a message.

4 Meta-model

The meta-model is a formalization of the component model. It models its concepts as classes and shows the relations among them. Following the different levels in Progress CM, the meta-model is also divided to two parts — System level and ProSave level.

4.1 System level

A system is at the system-level represented by class `System` (see Figure 10). A sub-system in a system is represented by class `SubsystemInstance`, which refers to a particular descendant of class `Subsystem` as its implementation. The distinction of the `SubsystemInstance` and `Subsystem` is to allow for instantiating the same sub-system inside a system several times.

Each sub-system is equiped by message ports (classes `InputMessagePort` and `OutputMessagePort`), which it uses for communication with other subsystems (see Figure 11).

The distinction between a sub-system and its instance impacts also ports. The class `MessagePort` and its descendants serve for defining the ports of a sub-system. A sub-system instance, however owns its own set of port instances (`MessagePortAttachmentPoint`), which are used in connecting the sub-system instances. The port instances of a sub-system instance must correspond one-to-one to the ports defined by the respective sub-system.

The communication between sub-system instances is realized by explicit message channels (class `MessageChannel`). The actual connection of a subsystem instance to a particular message channel is realized by class `Connection` which attaches the message channel to a port of the sub-system instance (class `MessagePortAttachmentPoint`).

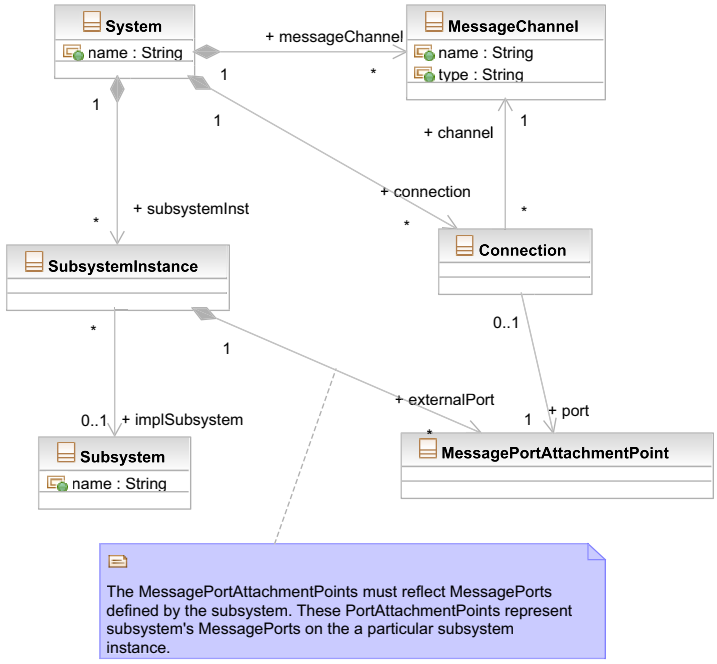


Figure 10: System level metamodel — Sub-systems

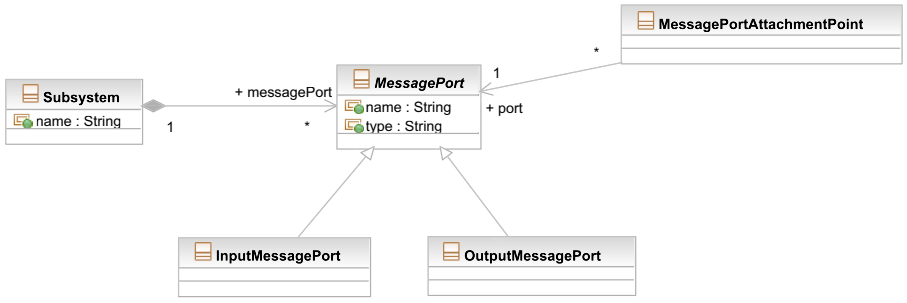


Figure 11: System level metamodel — Ports

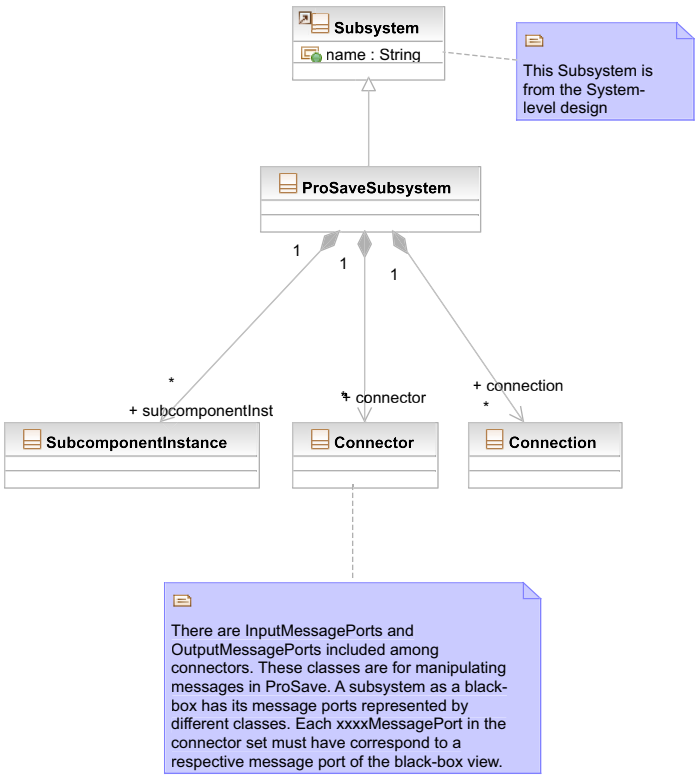


Figure 12: ProSave level metamodel — Subsystems

4.2 ProSave level

The top-level of a ProSave design is represented by class `ProSaveSubsystem` (see Figure 12) which is a specialization of the sub-system at the System level (class `SubSystem`).

The internals of a ProSave subsystem are modeled by sub-component instances, connectors and connections.

A sub-component instance (class `SubcomponentInstance`) represents a particular instantiation of a component (class `Component`). (This is similar to sub-systems and their instances at the system level.) The class `Component` is abstract and it has two specializations — the primitive component and the composite component.

By itself `Component` defines the services (class `Service`), each of which splits to an input port-group (class `InputPortGroup`) and a number of output port-groups (class `OutputPortGroup`). Each port-group defines one trigger port and a set of data ports (see Figure 13).

Ports are categorized and represented in the meta-model by classes `Input-`

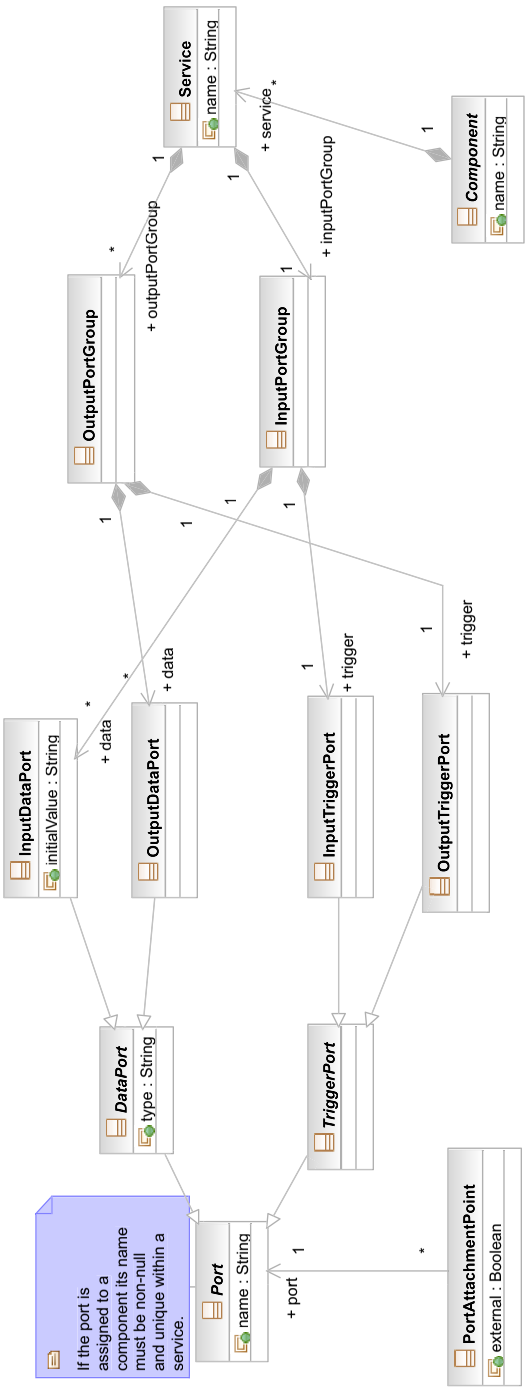


Figure 13: ProSave level metamodel — Services and ports

DataPort, OutputDataPort, InputTriggerPort, OutputTriggerPort and their abstract ancestors DataPort, TriggerPort, Port.

In a similar way to the system-level, there is also class PortAttachmentPoint representing a port of a sub-component instance (thus making possible to refer to a port of a particular sub-component instance).

The flow between sub-component instances is driven by connectors and connections (see Figure 14). There are a number of connectors defined in ProSave. Each type of a connector is represented by a dedicated class in the meta-model (DataFork, Selection, ControlJoin, etc.).

Each connector defines its ports by including descendants of Port. However, to allow for common handling of components and connectors, each connector defines also its set of port instances (MessagePortAttachmentPoint which correspond one-to-one to the connector ports.

The connectors at the top-level of ProSave include also message input/output ports (classes InputMessagePort and OutputMessagePort — see Figure 15). These correspond to ports defined by the sub-system and in fact make the sub-system ports accessible to the ProSave design.

The linkage between connectors and components is realized by connections. Each connection (class Connection) connects two port instances (PortAttachment-Point) together (see Figure 16).

The ProSave component model is nested, meaning that each component may be either primitive or composite.

A primitive component (class PrimitiveComponent) is tied to a particular implementation in C programming language (see Figure 17). It also provides mapping of each of its services to a particular C-method and for each service it defines mapping of ports to C-variables.

A composite component (class CompositeComponent) is modeled in a similar way as the ProSave subsytem on the top-level (see Figure 16) — by sub-component instances, connectors and connections. However, an important disction is that only a restricted subset of connectors may be used inside a composite component — only connectors inheriting from class ConnectorInsideComponent.

In addition to those, CompositeComponent has a set of port instances (Port-AttachmentPoint) which correspond one-to-one to its ports. These port instances are used to connect component internals to its external ports (i.e. to make delegations).

When using the PortAttachmentPoint for this purpose (i.e. internal port instances), the direction defined by the corresponding Port is used inverted (i.e. an input port becomes and output port and vice-versa). To mark this change, PortAttachmentPoint contains the property external which is set to false in this case.

Virtually any element in ProSave design may have a set of attributes (see Figure 18). These attributes capture requirements, models and other properties. On the level of a meta-model this is captured by abstract class Attribute, which should be specialized to model a particular requirement, quality attribute, timing information, etc.

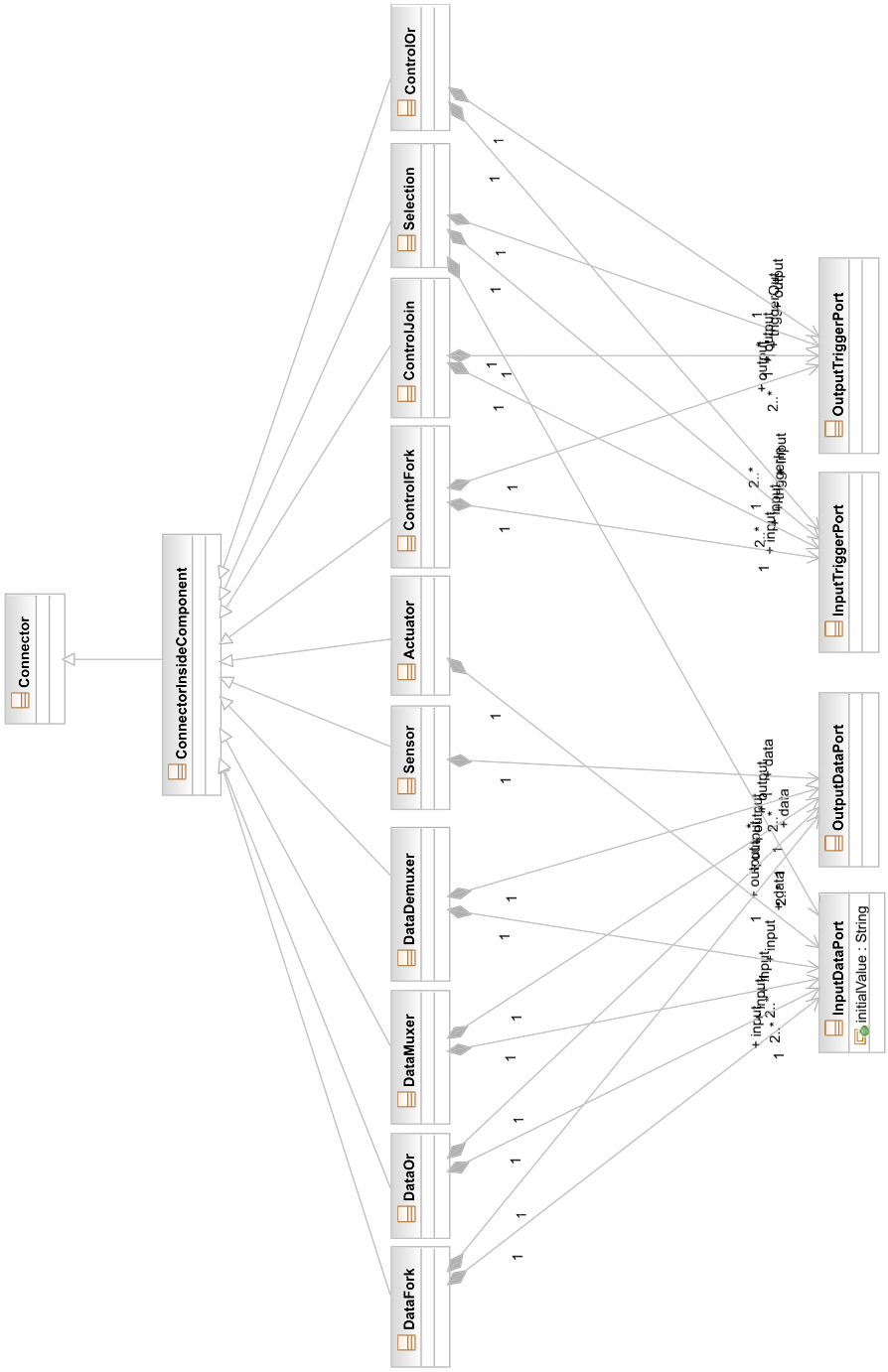


Figure 14: ProSave level metamodel — Connectors

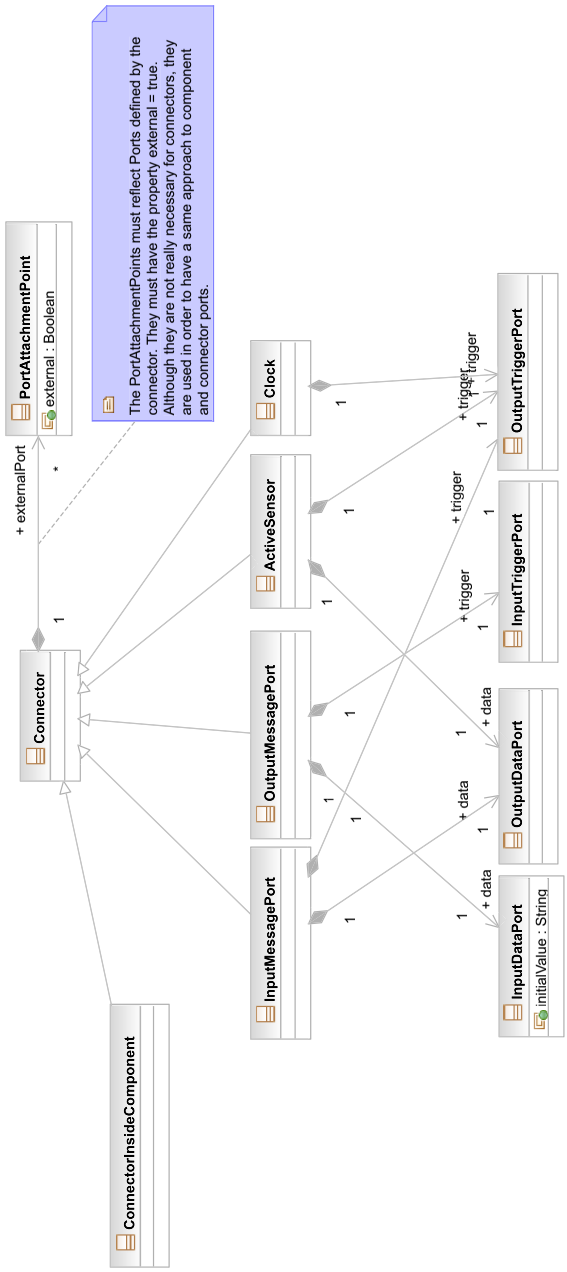


Figure 15: ProSave level metamodel — Additional top-level connectors

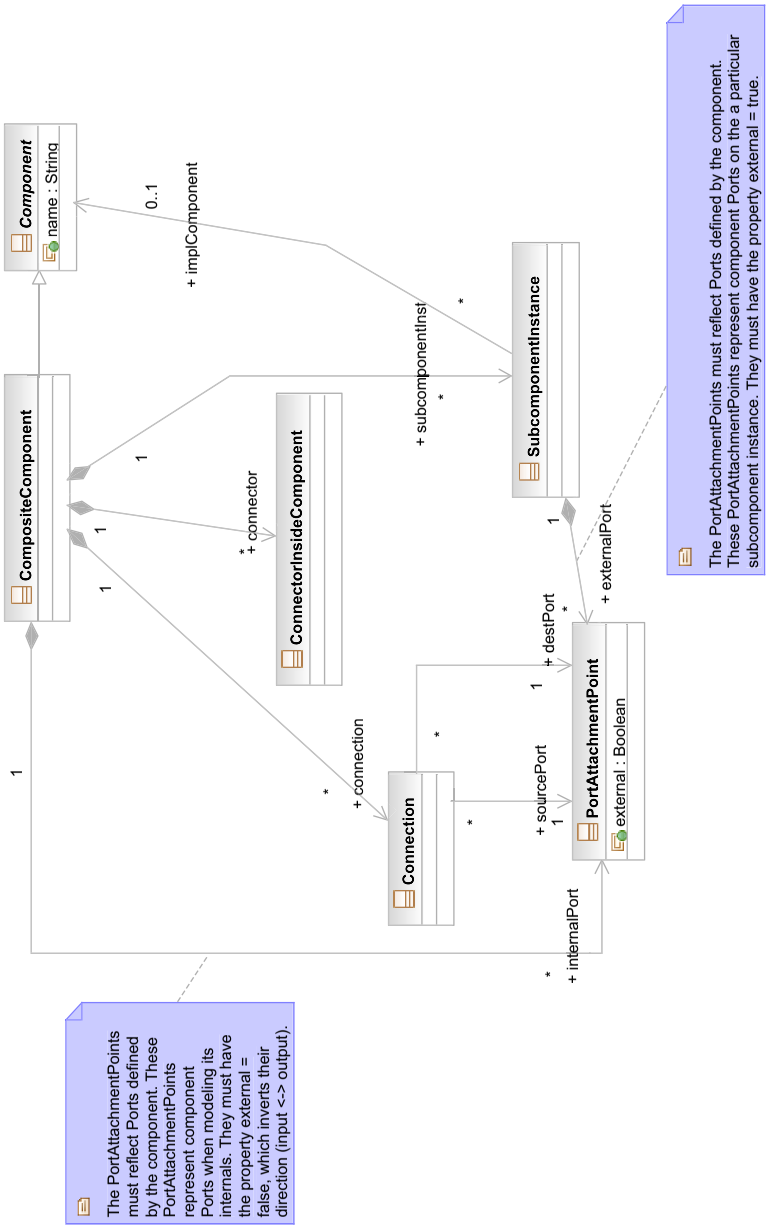


Figure 16: ProSave level metamodel — Composite components

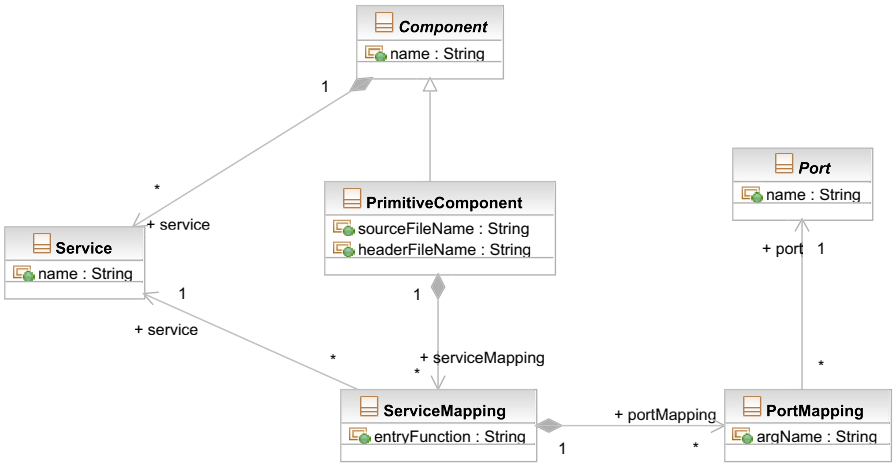


Figure 17: ProSave level metamodel — Primitive components

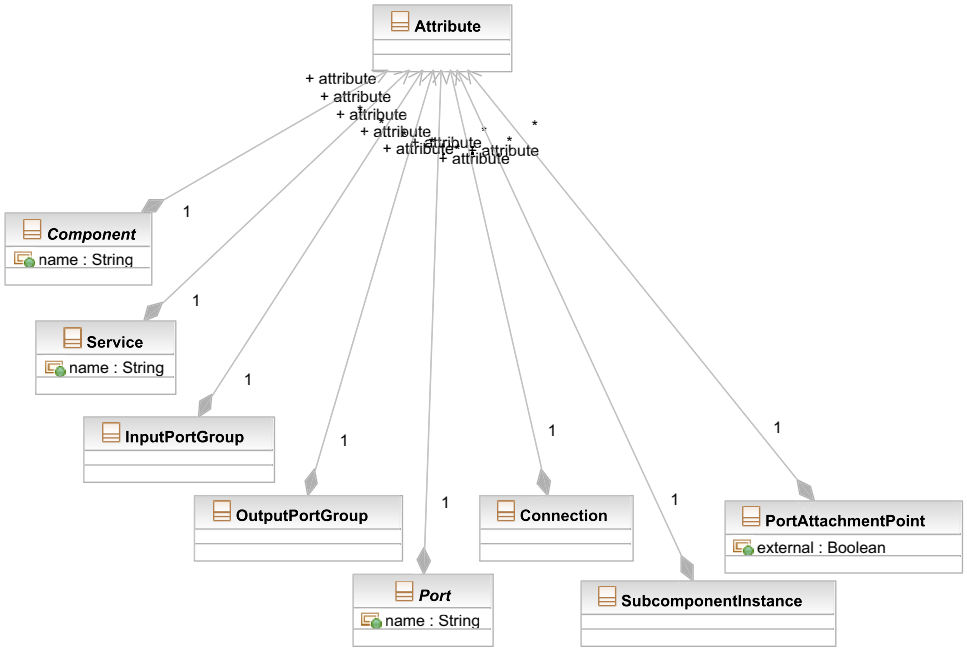


Figure 18: ProSave level metamodel — Component attributes

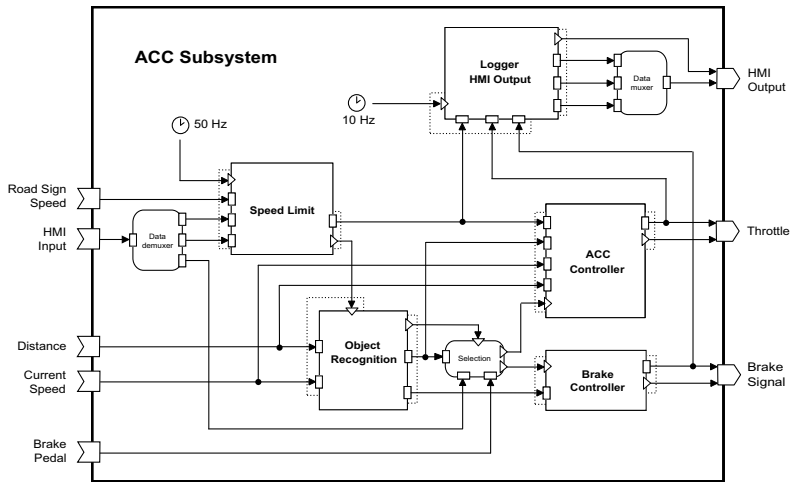


Figure 19: The design of the ACC subsystem.

5 Example

In order to illustrate the use of Progress CM we present a simple design of an Adaptive Cruise Controller (ACC) that was as well used as a case study throughout the development of the SaveComp component model [1].

The ACC works as a regular cruise controller if the road is free. But, if there is another vehicle in front moving with a lower speed, the ACC automatically reduces the vehicle's speed. If the ACC is enabled it also adjusts the maximum speed of the vehicle depending on the speed limit regulations.

The ACC itself forms a subsystem (as depicted in Figure 19). It communicates with other subsystems using its messages ports. From other subsystems it receives the information about the current traffic situation and input from the driver. Based on these values it computes braking force and throttle adjustment, which is communicated to the brake subsystem and to the engine control, respectively. Futher, it reports the state of the subsystem to the driver (HMI output).

The ACC subsystem is internally modelled using ProSave. It is a purely time-triggered system in the sense that it is not triggered by incoming messages — it uses only their data. The subsystem contains two activities triggered at different frequencies. The control functionality, which controls the throttle and brakes is triggered at 50 Hz. The logging and HMI output functionality is executed at a lower frequency of 10 Hz.

The control functionality executing at 50 Hz frequency can choose between two execution paths: braking (Brake controller component) or controlling the vehicle's throttle (ACC controller). In order to do this choice, a selection connector is used. The connector has one input trigger port coming from the Object

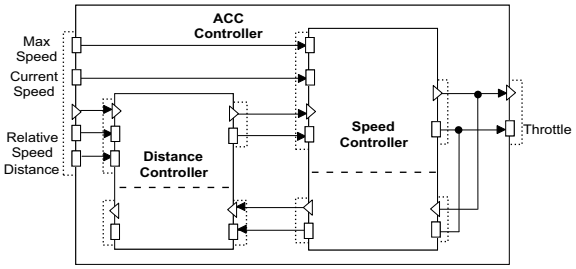


Figure 20: The design of "ACC Controller".

recognition component, three data input ports and two outputs trigger ports; one connected to the ACC controller and the other one to the Brake controller component. Further, a control-or connector is used to join the control flows of these two workflows.

The responsibilities of individual components are as follows:

Speed Limit component estimates the maximum speed of the vehicle depending on the speed limit regulations.

Object recognition component determines if there is a vehicle in front and in that case estimates the relative speed. If there is a need for braking it sends information to the Brake controller component.

Brake controller component is used to assist the driver if there is a vehicle in front and continuing at the same speed would lead to a collision.

ACC controller is a composite component (see Figure 20) consisting of a distance controller and a speed controller component (see Figure 21). It is used to regulate the throttle control of the vehicle on the basis of the current speed, desired speed and distance to the vehicle in front. Both, the distance controller component and the speed controller component have similar architecture. Each has two services – one corresponding to the control chain and the other to the feedback chain.

Logger HMI Output component gives information to the driver about the state of the vehicle and latest request.

References

[1] Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The SAVE approach to component-based development of vehicular systems. *Journal of Systems and Software*, 80(5):655–667, May 2007.

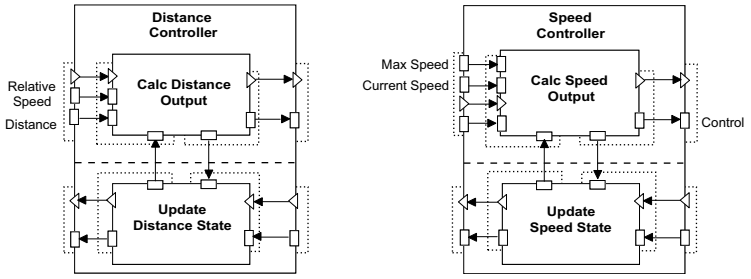


Figure 21: The design of "Distance Controller" and "Speed Controller".

[2] Hans Hansson, Ivica Crnković, and Thomas Nolte. The world according to PROGRESS. Internal document, November 2007.

[3] Kurt-Lennart Lundbäck, John Lundbäck, and Mats Lindberg. Development of dependable real-time applications. Arcticus Systems, December 2004.