ProCom — the Progress Component Model

Reference Manual

version 1.0

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

Mälardalen University, Västerås, Sweden

June 16, 2008

Contents

1 Introduction **3**
1.1 The PROGRESS approach 3
1.2 Component model overview 4

2 ProSys —the upper layer **5**
2.1 Subsystems 5
2.2 Connecting subsystems 5
2.3 Primitive and composite subsystems 6

3 ProSave — the lower layer **7**
3.1 Components 7
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 13
3.3.2 Composite components 13
3.4 Abstract execution semantics 14
3.5 Using ProSave in a ProSys subsystem 15

4 Example **17**

5 Meta-model **19**
5.1 ProSys package 19
5.2 ProSave package 20

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 domains of vehicular systems, automation and telecom.

The component model is influenced by previous work in the SAVE project[2] and also to some extent by the Rubus component technology [2].

1.1 The Progress approach

The goal of PROGRESS is to provide theories, methods and tools to increase quality and reduce costs in the development of systems in the vehicular-, automation- and telecommunication domains. Together they are to cover the whole development process, supporting the consideration of predictability and safety throughout the development. As a basis for this work, PROGRESS proposes a component-based development approach.

To be able to apply component-based strategy throughout the development process (starting from a vague specification of the system based on early requirements up to its final and precise specification and implementation ready to be deployed), PROGRESS adopts a concept of a component as a collection gathering all the information needed and/or specified at different points of time of the development process. That means a component comprises requirements, documentation, source code, various models (e.g. behavioural and timing), predicted and experimentally measured values (e.g. performance and memory consumption), etc.

To achieve predictability, PROGRESS puts a strong emphasis on analysis to provide estimations and guarantees of different important properties. Analysis is present throughout the whole development process and gives results depending on the completeness and accuracy of the models and descriptions. Early, inexact analysis may be performed during design to guide design decisions and provide early estimates. Once the development is completed, analysis may be used to validate that the created components and their composition meet the original requirements. The different analyses planned for PROGRESS include reliability predictions, analysis of functional compliance (e.g. ensuring compatibility of interconnected interfaces), timing analysis (analysis of high-level timing as well as low-level worst-case execution time analysis) and resource usage analysis (e.g. memory, communication bandwidth).

The design and development of components is supplemented by deployment activities consisting of two parts: (1) allocation of components to physical nodes and (2) code synthesis. In code synthesis, the code of components are merged, optimised and mapped to artefacts of an underlying real-time operating system

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

(e.g., tasks). The binary images resulting from code synthesis are ready to be executed at the target physical nodes.

Separating component development and deployment facilitates reusing components over different physical architectures. However, since much of the desired analysis requires information about allocation and the underlying platform, the component model and the deployment models must be related.

The overall development approach envisioned by PROGRESS is further elaborated in the concept paper [3].

1.2 Component model overview

The kind of complex distributed embedded systems found in the targeted domains typically have quite different characteristics when considered at different levels of granularity. The big parts of the system are different from the small parts of some low-level control task, in terms of execution model, communication style, synchronisation, etc., but also with respect to the kind of information that must be available and the type of analysis that is appropriate.

In general, systems consist of relatively independent big units encapsulating complex functionality. In a distributed system, much of the communication between and within these big units manifests as messages sent over a bus in order to share data or to notify other parts of the system important events. At a more fine-grained level, parts of the detailed control functionality can be provided by small units that tend to have dedicated, restricted functionality, simpler communication often manifesting within a single physical node, and stronger timing and synchronisation requirements.

To address the different concerns at different levels of granularity, ProCom consists of two distinct, but related, layers [1]. At the upper layer, called ProSys, the system is modelled as a number of active and concurrent subsystems, communicating by message passing. The lower layer, ProSave, addresses the internal design of a subsystem down to primitive functional components implemented by code. Contrasting ProSys subsystems, ProSave components are passive and the communication between them is based on a pipes-and-filters paradigm.

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 ProSys and ProSave layers are described in Section 2 and Section 3, respectively, and Section 3.5 define the relation between them. Section 4 presents an example of modelling a particular subsystem. Section 5 describes the metamodel, formalising the concepts of the component model and specifying how they relate.

2 ProSys —the upper layer

In ProSys, a system is modelled as a collection of concurrent, communicating *subsystems*. A subsystem can in turn be built out of smaller subsystems, thus making ProSys a hierarchical component model. From the perspective of component-based development, subsystems are the "components" conforming to the ProSys component model, defined as design or implementation units that can be developed independently, stored in a repository and reused in multiple applications.

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 Subsystems

A subsystem is specified by typed input- and output *message ports*, as shown in Figure 1. The ports express what messages the subsystem receives and sends. The external view also includes attributes and models.

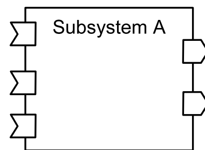


Figure 1: External view of a subsystem with three input message ports and two output message ports.

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.

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* (see Figure 2). This means that information about the shared information can be associated with the channel, rather than with the producer or consumer. It also makes it possible to define that a particular data, e.g., vehicle speed, will be required in the system before the producer and receivers of this data are defined. 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.

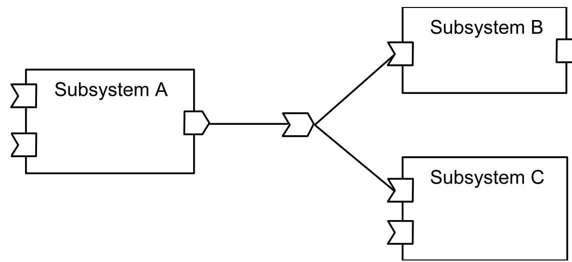


Figure 2: Three subsystems communicating via a message channel.

Message passing is asynchronous, meaning that sending a message is a non-blocking action. The way in which new messages are handled is defined by the receiving subsystem (e.g., if the input message port is polled at a fixed frequency or if the arrival of a new message directly triggers a response, if additional messages are queued, ignored or overwrited previous messages, etc.).

2.3 Primitive and composite subsystems

Primitive subsystem can be internally modelled by ProSave components, as described in Section 3.5. Alternatively, they can be realised by code conforming to the runtime interface of Progress subsystems[3]. 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 [3].

A composite subsystem internally consists of subsystems and message channels. There are also connections that associate the 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 acts 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 seen as a way to "unify" a channel defined locally within a composite component with a particular channel in the component environment[4].

An example of a composite subsystem is given in Figure 3.

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

[4]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.

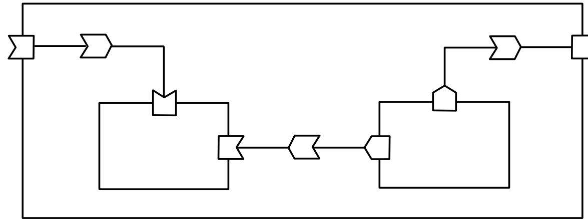


Figure 3: An example of a composite subsystem.

3 ProSave — the lower layer

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 ProSys.

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 3.5.

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 architectural style is based on a data/control flow model with an explicit separation of data transfer and control flow. The 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, primarily designed to encapsulate relatively small and rather low-level, non-distributed functionality. 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. The black-box view of a component, based only on the externally visible

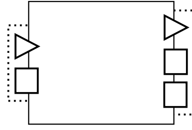


Figure 4: A simple ProSave component with two input ports to the left and three output ports to the right. Triangles and boxes denote trigger ports and data ports, respectively.

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 white-box view of components, where the contents of a component is available, e.g., the sub-component structure or the source code. 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.

Figure 4 shows a component in its simplest form. When the input trigger port is activated, it reads the current value at the input data port and starts processing this value. When done, output is produced at the output data ports on its right side, and control is forwarded via the output trigger port. The general component interface, defined in detail below, allows components to provide more than one service, and to produce parts of the output at different points in time.

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 functionality that the component provides (e.g. control loop, diagnostics). Services are triggered independently and they may be run concurrently. 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 referred 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.[5]

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

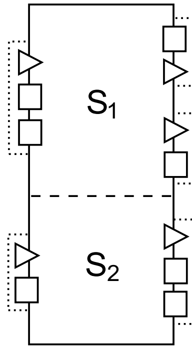


Figure 5: External view of a component with two services; S_1 has two output groups and S_2 has a single output group.

Allowing multiple output groups provides the possibility to produce outputs at different points in time.

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, 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 receive 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.[6]

While in the running state, a service can not be triggered again (i.e., activations of the input trigger port are simply ignored). To avoid inefficiency, we envision that this is ensured by analysis at design time, rather than enforced by a runtime mechanism.

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

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

by additional *connectors* providing more elaborate manipulation of the data- 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 (see Section 3.3.2) can have one connection on the inside and one on the outside.[7]

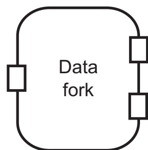
A connection between data ports denotes the data transfer. ProSave follows the push-model for data transfers. It means that whenever data is produced on a data output port, the data is transferred by the connection to the input data port and stored there, overwriting the previous value. A triggered component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

Connections between trigger ports define 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 3.4. This exception essentially specifies that when data and triggering appear together at an output port group, the data should be delivered before the triggering.

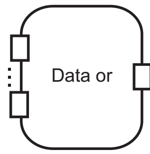
3.2.2 Connectors

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.

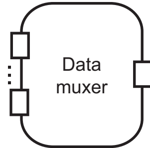


A *data fork* is used to split a data connection to several ones. It has one input data port and at least two output data ports. Data written to the input port are duplicated on the output ports. Graphically, this connector can also be denoted by a thick dot.

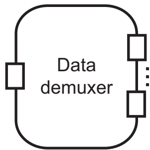
[7]Strictly speaking, we should say one connection on the inside of the component and one on the outside of each *instance* of the component (see Section 5).



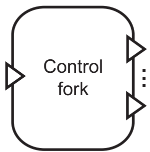
The *data or* connector is used to merge several data connections to one. It has one output data port and least two input data ports. Data written to any of the input ports are forwarded to the output port.



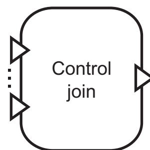
A *data muxer* allows grouping several data inputs into one output. It is mainly used to build data of a message (exchanged at a system level). It has two or more input data ports and one output data port. The type of the output data port is a struct comprising data of all input data ports. Whenever 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.



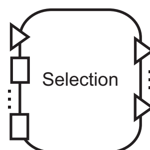
A *data demuxer* works as an inverse to data muxer. It has two or more output data ports and one input data port. Whenever data is written to the input data port, it is extracted and respective parts made available on the output data ports.



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. The alternative notation for this connector is a small circle.



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.

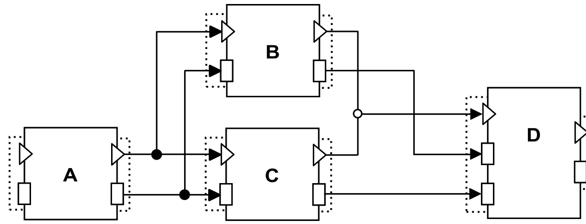


Figure 6: A typical usage of fork- and join connectors. When component A is finished, components B and C are executed in arbitrary order (possibly interleaved). Component D is executed once both B and C have finished.

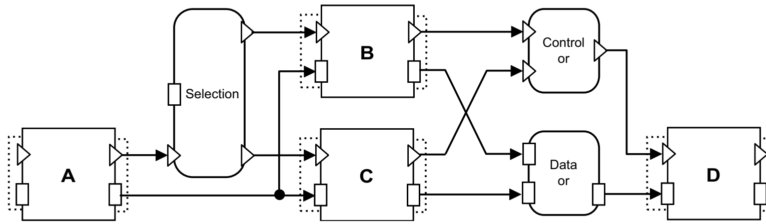
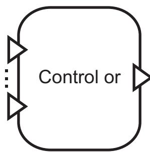


Figure 7: A typical usage of selection and or-connectors. When component A is finished, either B or C is executed, depending on the value at the selection data port. In either case, component D is executed afterwards, with the data produced by B or C as input.



The *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.

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.

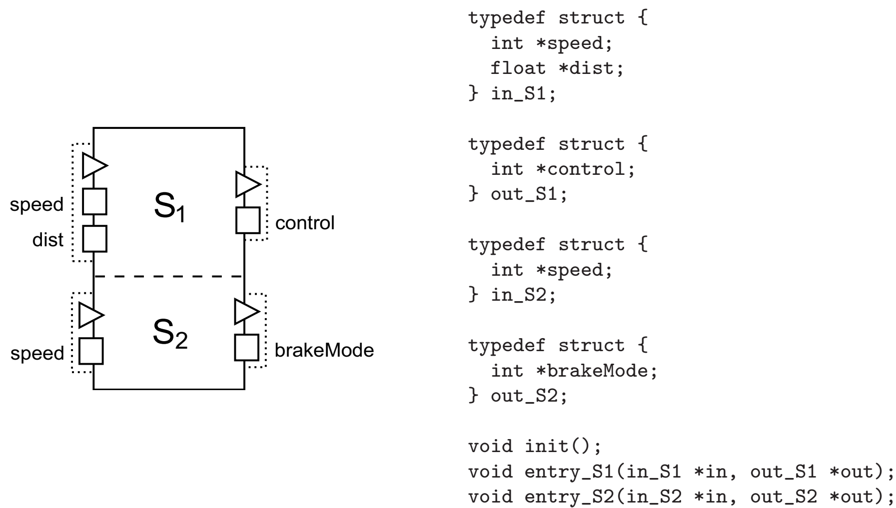


Figure 8: Example of header file for a primitive component with two services, and no explicit name mappings.

3.3.1 Primitive components

In a primitive component, the behaviour of each service is realised by a non-suspending C function. In addition, the component 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[8]. 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 Section 5.2). Figure 8 shows an example of a header file.

3.3.2 Composite components

The internal view of a composite component consists of sub-components, connections and connectors. Each sub-component is, in turn, an instance of 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

[8]The current version of the API is restricted to primitive components with at most one output group per service

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 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. 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.

The overall responsibility of a composite component is to realise the internal workflows defined by connections, connectors and subcomponents. Concretely, this amounts to transferring 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 trigger signal should not arrive to any port before all data have arrived to all 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 output 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 (i.e., outside the enclosing component), 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.

3.5 Using ProSave in a ProSys subsystem

Components in ProSave are passive, typically local and they communicate via data exchange and triggering. A ProSys 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.

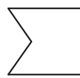
Moreover, subsystems use message passing with explicit message channels as the means of communication, and parts of a single subsystem often end up on different nodes in the final system.

This section describes how ProSave can be used to define the internals of a ProSys 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 to allow for:

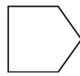
a) mapping between message passing and trigger/data communication, and

b) specifying activation of ProSave components.

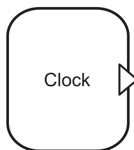
The additional connectors are described in detail in the list below.



An *input message port* of the subsystem acts as a ProSave connector with 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.



The *output message port* is an inverse to the input message port. In ProSave it acts as a connector with 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 ProSys is performed only at the 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 connector with a trigger and a data port, which can be connected to other components or connectors in the ordinary ProSave way. Additionally, clocks are used to generate periodic triggers.

The connections on the top level inside a subsystem follow the same semantics as the connections inside the ProSave component. A transfer activity is initiated when data or triggering appears at an output port of a component

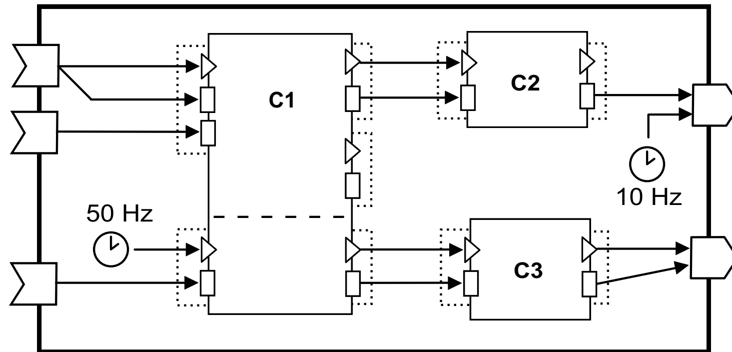


Figure 9: A ProSys subsystem internally modelled by ProSave.

or 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 Example

To illustrate the component model we use as an example an electronic stability control (ESC) system from the vehicular domain. In addition to anti-lock braking (ABS) and traction control (TCS), which aim at preventing the wheels from locking or spinning when braking or accelerating, respectively, the ESC also handles sliding caused by under- or oversteering. Basically, when the ESC detects that the direction of the car differs significantly from the desired direction indicated by the steering wheel, it brakes one of the wheels to counteract the sliding.

When the car is modelled in ProCom, the ESC would be one of many subsystems at the top level, together with subsystems for engine control, airbags, climate control, etc. To simplify the example, we assume that none of the inputs to the ESC, i.e., yaw, acceleration and steering wheel position and wheel speed, are needed by any other subsystem. We also assume that the ESC does not need any additional information from other subsystems, ignoring for example any specification of how the ESC should react if some other part of the system is not functioning properly.

For the output, we assume that the ESC has direct access to actuators for adjusting the brake pressure of individual wheels. Acceleration adjustments, however, are output as messages to the engine subsystem. The ESC also produces messages indicating if it is active or not, used for example by the information panel to warn the driver.

The ESC can be further decomposed, as shown in Figure 10. Inside, we find subsystems for the sensors and actuators that are local to the ESC. There are also subsystems corresponding to specific parts of the ESC functionality (SCS, TCS and ABS). In the envisioned scenario, the TCS and ABS subsystems are

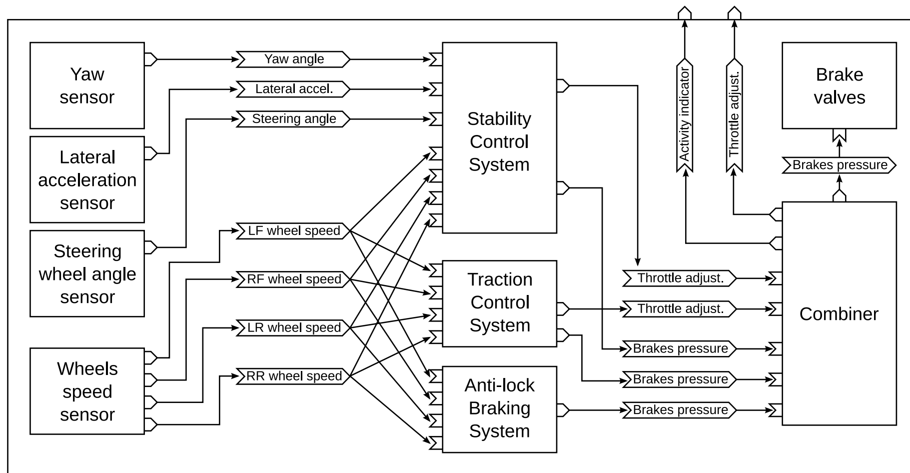


Figure 10: The ESC is a composite subsystem, internally modelled by local message channels and smaller subsystems.

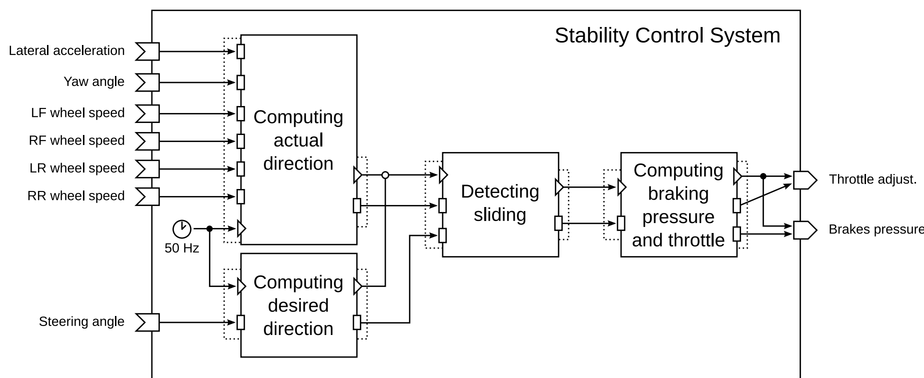


Figure 11: The SCS subsystem, modelled in ProSave.

reused from previous versions of the car, while SCS corresponds to the added functionality for handling under- and oversteering. Finally, the "Combiner" subsystem is responsible for combining the output of the three. The contents of the SCS subsystem are further elaborated in the following section.

To exemplify the ProSave level of the component model, the contents of the SCS subsystem is shown in Figure 11. SCS is a primitive subsystem, internally modelled in ProSave. It contains a single periodic activity, performed at a frequency of 50 Hz. This activity first activates the components responsible for computing the actual and desired directions, respectively. When both have finished, another component compares the two directions and decides whether

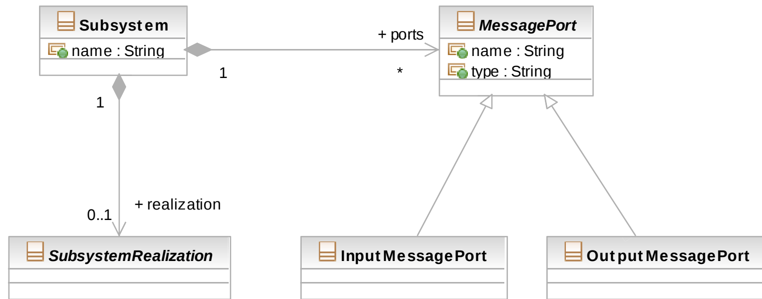


Figure 12: ProSys metamodel — Subsystem

or not the stability control should be activated. The actual response, i.e., the adjustment of brakeage and acceleration, is computed by a fourth component.

5 Meta-model

The meta-model is a feature of the component model. It models its concepts as classes and shows the relations among them. Following the different levels in ProCom, the meta-model is also divided to two packages — ProSys and ProSave.

5.1 ProSys package

A subsystem is represented by a class `Subsystem` (see Figure 12). Each subsystem has message ports (classes `InputMessagePort` and `OutputMessagePort`), which it uses for communication with other subsystems. The top-level system is in ProSys viewed as a special subsystem (typically with no message ports) and thus it is also represented by class `Subsystem`.

The class `Subsystem` refers to a realisation of the subsystem, which may define the subsystem as a composition of other subsystems or as a composition of ProSave components. If the realisation of a subsystem is undefined, the class `Subsystem` provides only a black-box view of a subsystem, which is advantageous in early stages of design when the realisation of a subsystem is not decided yet.

The subsystem realisation that defines the subsystem as a composition of other subsystems is captured by class `CompositeSubsystem` (see Figure 13). It lists internal subsystem instances, message channels and connections. An internal subsystem instance (class `SubsystemInstance`) refers to a Subsystem that implements it. In early stages of design, this reference may be not set, in which case the subsystem instance serves as a placeholder to be assigned the implementation later.

The communication within a subsystem is realised by explicit message channels (class `MessageChannel`). The connection of between a message channel

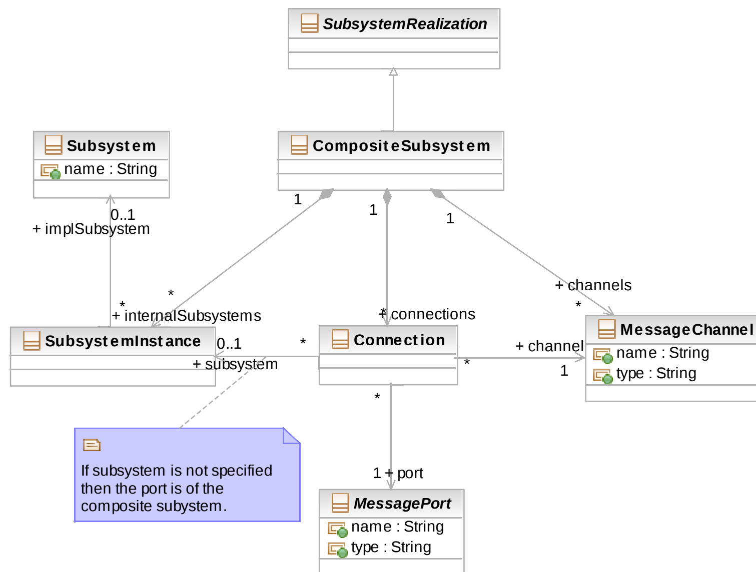


Figure 13: ProSys — Composite Subsystem

and a subsystem's port is realised by class `Connection`. It connects the message channel to a port of a subsystem instance (specified by referring to the subsystem instance and the port of the respective subsystem) or it delegates messages between the message channel and a port of the composite subsystem (the reference to subsystem instance is left unset in this case).

5.2 ProSave package

The top-level of a ProSave design is represented by class `ProSaveSubsystem` (see Figure 14) which acts as a realisation of a ProSys subsystem.

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

A subcomponent instance (class `SubcomponentInstance`) represents a particular instantiation of a component (class `Component`). (This is similar to subsystems and their instances within a composite subsystem.)

The class `Component` (see Figure 15) 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.

Ports are categorized and represented in the meta-model by classes `InputDataPort`, `OutputDataPort`, `InputTriggerPort`, `OutputTriggerPort` and their abstract ancestors `DataPort`, `TriggerPort`, `Port`.

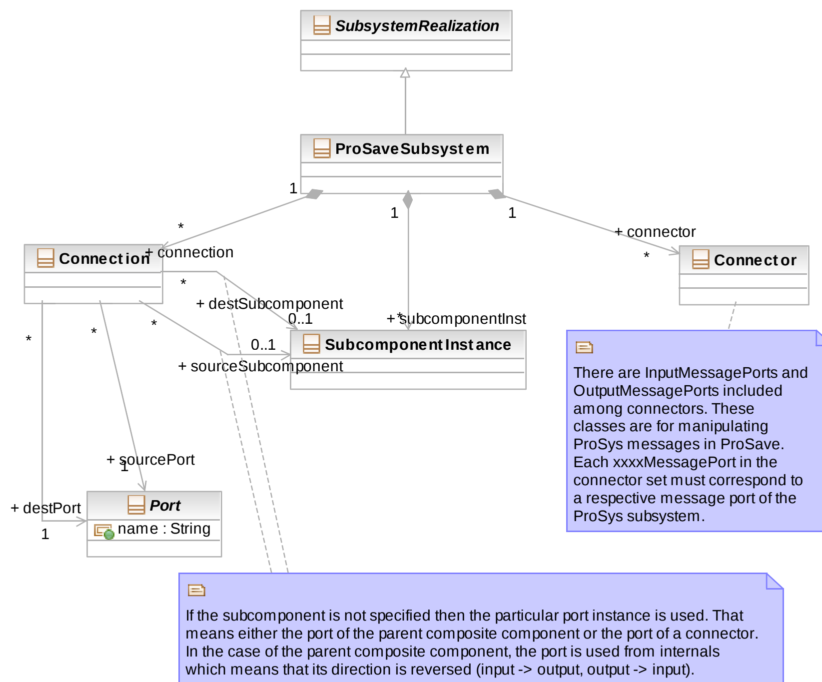


Figure 14: ProSave — Subsystem

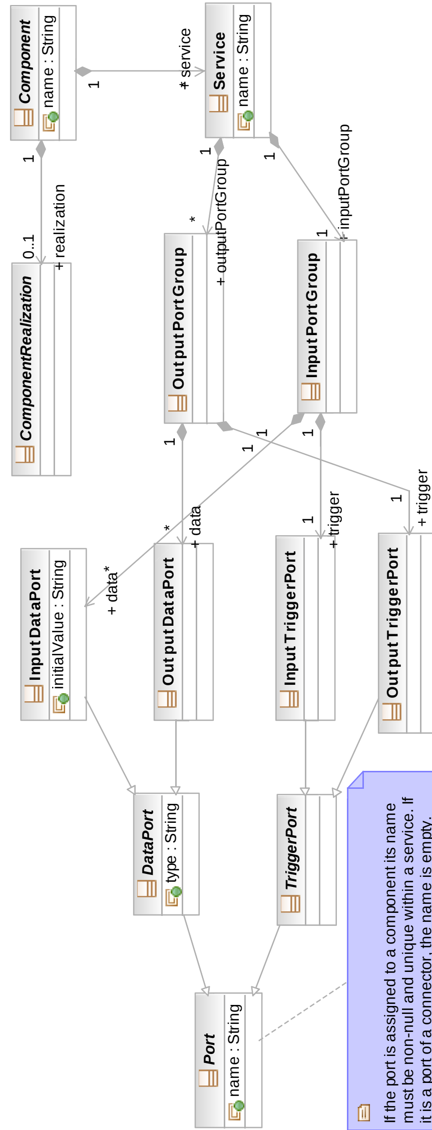


Figure 15: ProSave — Component, Services and Ports

The flow between subcomponent instances is driven by connectors and connections (see Figure 16). 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`.

The connectors at the top-level of ProSave include also message input/output ports (classes `InputMessagePort` and `OutputMessagePort` — see Figure 17). These correspond to ports defined by the subsystem and in fact make the subsystem message ports accessible to the ProSave design.

The linkage between connectors and/or components is realised by connections. A connection (class `Connection`) connects two port ports together. Similarly to connections between subsystems, the connection may refer to a subcomponent instance, in which case the port is related to that particular instance. Otherwise, the port is used directly, which is the case of ports used in delegation and the ports of connectors.

A component contains a link to its realisation (class `ComponentRealization`). Similarly to subsystems, this feature allows having only black-box components at early stages of design and assigning them the realisation later on. ProSave distinguishes two realisations — either by a composition of subcomponents (i.e. composite component) or by code (i.e. primitive component).

A composite component (class `CompositeComponent`) is modeled in a similar way as the ProSave subsystem on the top-level (see Figure 18) — by subcomponent instances, connectors and connections. However, an important distinction is that only a restricted subset of connectors may be used inside a composite component — only connectors inheriting from class `ConnectorInsideComponent`.

A primitive component (class `PrimitiveComponent`) is tied to a particular implementation in C programming language (see Figure 19). 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.

Virtually any element in ProSave design may have a set of attributes (see Figure 20). 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 specialised to model a particular requirement, quality attribute, timing requirement, etc.

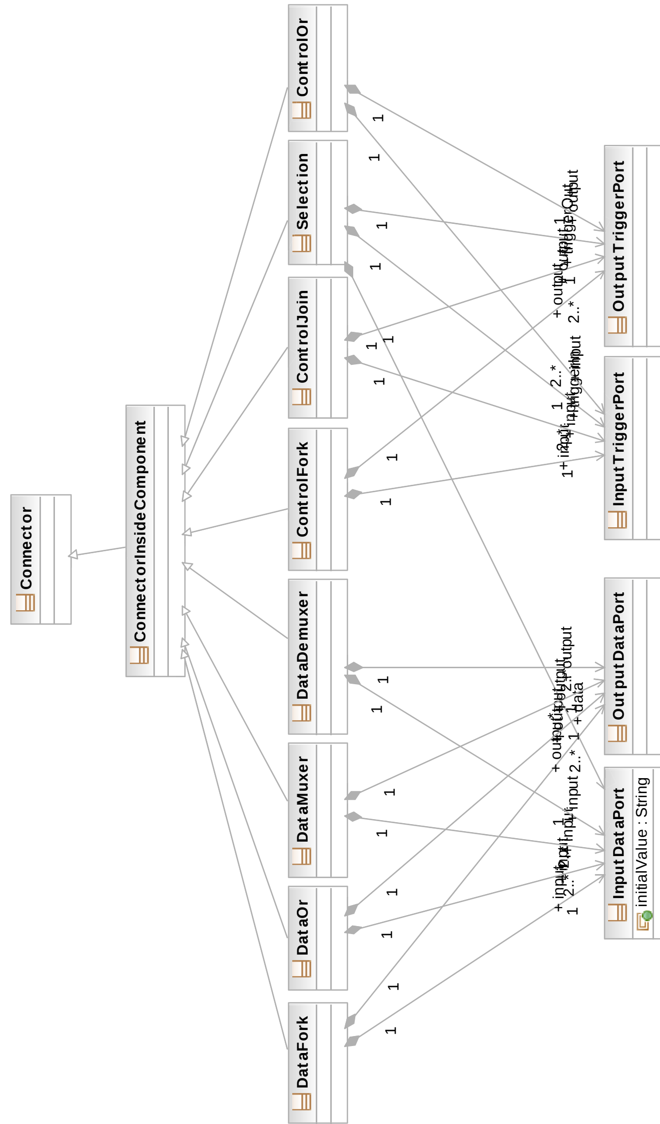


Figure 16: ProSave — Connectors

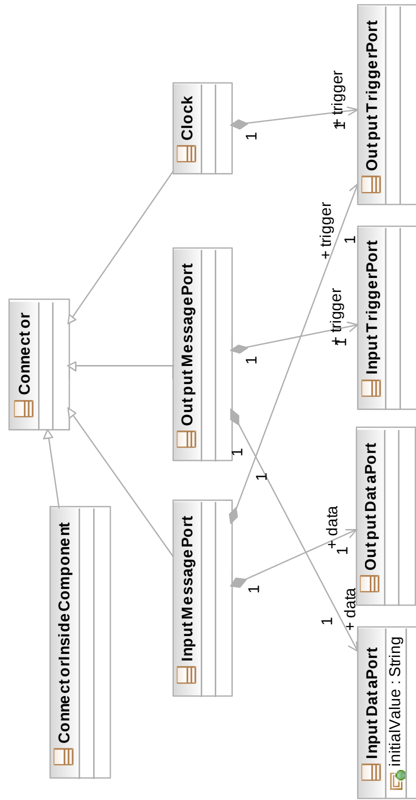


Figure 17: ProSave — Additional Top-Level Connectors

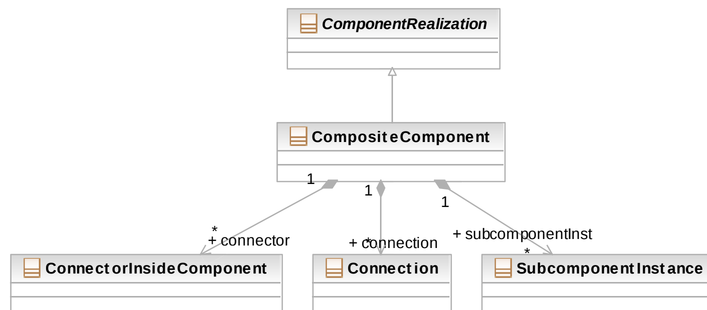


Figure 18: ProSave — Composite Component

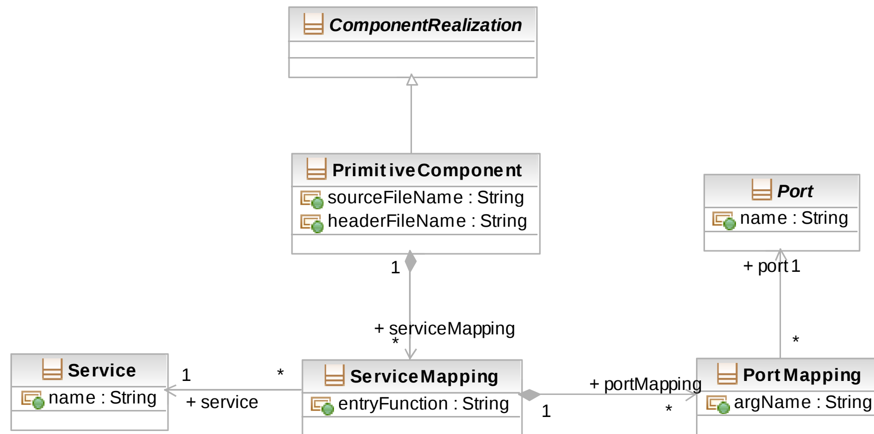


Figure 19: ProSave — Primitive Component

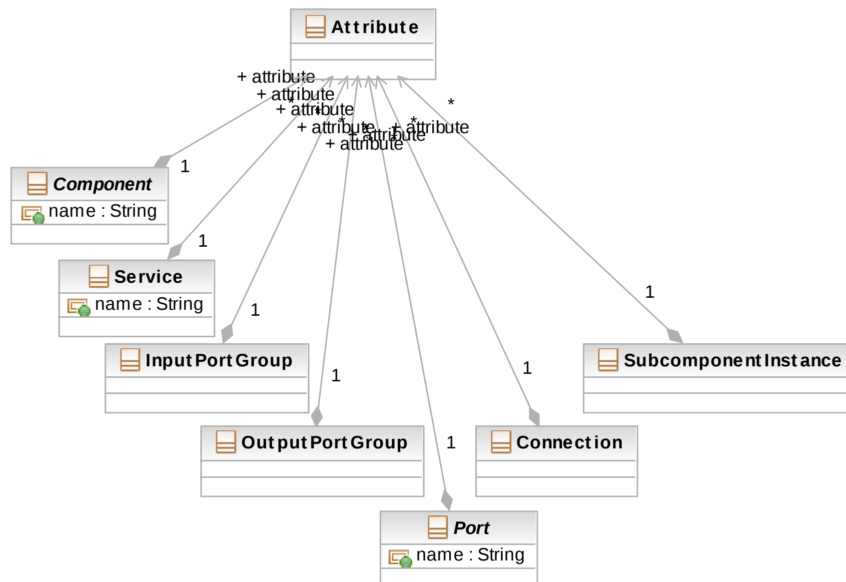


Figure 20: ProSave — Component Attributes

References

[1] Tomáš Bureš, Jan Carlson, Séverine Sentilles, and Aneta Vulgarakis. A component model family for vehicular embedded systems. In *The Third International Conference on Software Engineering Advances*, October 2008.

[2] Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, Mats Lindberg, John Lundbäck, and Kurt-Lennart Lundbäck. The Rubus component model for resource constrained real-time systems. In *3rd IEEE International Symposium on Industrial Embedded Systems*, Montpellier, France, June 2008.

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