# SaveCCM – a component model for safety-critical real-time systems

Hans Hansson[1], Mikael Åkerholm[1], Ivica Crnkovic[1], Martin Törngren[2]
[1]*Mälardalen Real-Time Research Centre, http://www.mrtc.mdh.se*
*Mälardalen University, Västerås, Sweden*
*{hans.hansson, mikael.akerholm, ivica.crnkovic}@mdh.se*
[2]*Mechatronics Division, Dept. of Machine Design, KTH, Stockholm, martint@kth.se*

## Abstract

*Component-based development has proven effective in many engineering domains, and several general component technologies are available. Most of these are focused on providing an efficient software-engineering process. However for the majority of embedded systems, run-time efficiency and prediction of system behaviour are as important as process efficiency. This calls for specialized technologies. There is even a need for further specialized technologies adapted to different types of embedded systems, due to the heterogeneity of the domain and the close relation between the software and the often very application specific system.*

*This paper presents the SaveCCM component model, intended for embedded control applications in vehicular systems. SaveCCM is a simple model in which flexibility is limited to facilitate analysis of real-time and dependability. We present and motivate the model, and provide examples of its use.*

## 1. Introduction

Component-based development (CBD) is of great interest to the software engineering community and has achieved considerable success in many engineering domains. Some of the main advantages of CBD are reusability, higher abstraction level and separation of the system development process from the component development process. CBD has been extensively used for several years in desktop environments, office applications, e-business and in Internet- and web-based distributed applications. The component technologies used in these domains originates from object-oriented (OO) techniques. The basic principles of the OO approach, such as encapsulation and class specification, have been further extended; the importance of component interfaces has increased: a component interface is treated as a component specification and the component implementation is treated as a black box. A component interface is also the means of integrating the components in an assembly. Component technologies include the support of component deployment into a system through the component interface. On the other hand, the management of components' quality attributes has not been supported by these technologies. In the domains in which these technologies are widely used, the quality attributes have not been of primary interest and have not been explicitly addressed; they have instead been treated separately from the applied component-based technologies. In many other domains, for example embedded systems, CBD is utilized to a lesser degree for a number of different reasons, although the approach is as attractive here as in other domains. One reason for the limited use of CBD in the embedded systems domain is the difficulty to transfer existing technologies to this domain, due to the difference in system constraints. Another important reason is the inability of component-based technologies to deal with quality attributes as required in these domains. For embedded systems, a number of quality attributes are at least as important as the provided functionality, and the development efforts related to them are most often greater than the efforts related to the implementation of particular functions. For development of vehicular systems, CBD is an attractive approach, but due to specific requirements of system properties such as real-time, reliability and safety, restricted resource consumption (e.g., memory and CPU), general-purpose component models cannot be used. Instead new component models that keep the main principles of the CBD approach, but fulfil specific requirements of the domain, must be developed.

This paper discusses the component model SaveCCM, a part of SAVEComp, a component-based development framework being developed in the project SAVE (Component Based Design of Safety Critical Vehicular Systems). The basic idea of SAVEComp is to by focusing on simplicity and analysability of real-time and dependability quality attributes provide efficient support for designing and implementing embedded control applications for vehicular systems.

This is the first paper on SaveCCM, focusing on structural issues, triggering of components, and timing behaviour. Our aim is to also consider dependability, though this is not the focus here.

The paper is organised as follows. Section 2 gives a short overview of different component models used in embedded systems. Section 3 briefly presents the SAVE project, and Section 4 outlines the characteristics of the considered application domain. In Section 5, our component model SaveCCM is presented, including textual and graphical syntax, as well as a few illustrative examples. A larger and more complete example from the vehicular domain is provided in Section 6, and in Section 7 we summarize and give an outline of future work.

## 2. Related work

In addition to widely used component technologies, new component technologies appear in different application domains, both in industry and academia. We will refer to some of them: Koala and Rubus used in industry and the research technologies PECT, PECOS and ROBOCOP.

The Koala component technology [9] is designed and used by Philips for development of software in consumer electronics. Koala has passive components that interact through a pipes-and-filters model, which is allocated to active threads. However, Koala does not support analysis of run-time properties.

The Robocop component technology [Jon03] is a variant of the Koala component technology. A Robocop component is a set of models, each of which provides a particular type of information about the component. An example of such a model is the non-functional model that includes modeling timeliness, reliability, memory use, etc. Robocop aims to cover all aspects of a component-based development process for embedded systems.

The Rubus Component Model [8] is developed by Arcticus systems aimed for small embedded systems. It is used by Volvo Construction Equipment. The component technology incorporates tools, e.g. a scheduler and a graphical tool for application design, and it is tailored for resource constrained systems with real-time requirements. In many aspects Rubus Component Model is similar to SaveCCM; actually some of the basic approaches from Rubus are included in SAVEComp. One difference is that SAVEComp is focused on multiple quality attributes and in-dependences of underlying operating system.

PECT (Prediction-enabled Component Technology) from Software Engineering Institute at CMU [12] [13] focuses on quality attributes specification and methods for prediction of quality attributes on system level from attributes of components. The component model enables description of some real-time attributes. Compared with SAVEComp, PECT is a more general-purpose component technology and more complex.

PECOS (PErvasive COmponent Systems) [6], developed by ABB Corporate Research Centre and academia, is designed for field devices, i.e. reactive embedded systems that gathers and analyze data via sensors and react by controlling actuators, valves, motors etc. The focus is on nonfunctional properties such as memory consumption and timeliness, which makes PECOS goals similar to SaveCCM.

These examples show that there are many similar component technologies for development of embedded systems. One could ask if it would not be more efficient to use a single model. Experiences have shown that for many embedded system domains efficiency in run-time resources consumption and prediction of system behavior are far more important than efficiency in the software development. This calls for specialization, not generalization. Another argument for specialization is the typically very close relation between software and the system in which the software is embedded. Different platforms and different system architectures require different solutions on the infrastructure and interoperability level, which leads to different requirements for component models. Also the nature of embedded software limits the possibilities of interoperability between different systems. Despite the importance of pervasiveness, dynamic configurations of interoperation between systems, etc. this is still not the main focus of vast majorities of embedded systems.

These are the reasons why different application domains call for different component models, which may follow the same basic principles of component-based software engineering, but may be different in implementations. With that in mind we can strongly motivate a need for a component technology adjusted for vehicular systems.

## 3. The SAVE project

The long term aim of the SAVE [10] project is to establish an engineering discipline for systematic development of component-based software for safety critical embedded systems. SAVE is addressing the above challenge by developing a general technology for component-based development of safety-critical vehicular systems, including

- Methodology and process for development of systems with components
- Component specification and composition, providing a component model which includes the basic characteristics of safety-critical components and infrastructure supporting component collaboration.
- Techniques for analysis and verification of functional correctness, real-time behaviour, safety, and reliability.
- Run-time and configuration support, including support for assembling components into systems, run-time monitoring, and evaluation of alternative configurations.

The main objective of SAVE is to develop SAVEComp – a component-based development (CBD) technology for safety-critical embedded real-time systems (RTS). The primary focus is on designing systems with components, based on component and system models. The ambition is to develop a method and infrastructure for CBD for safety-critical embedded RTS, corresponding to existing general component technologies, such as COM and JavaBeans.

## 4. Application characteristics

As mentioned above, the considered application domain is vehicular systems. Within that domain we are mainly considering the safety-critical sub-systems responsible for controlling the vehicle dynamics, including power-train, steering, braking, etc.

The vehicular industry has a long tradition of building systems from components provided by different suppliers. In the past these components have been purely mechanical, but today many of the components include computers and software. The trend today is, on one hand, towards "intelligent" mechatronics "light weight nodes", such as actuators including a microprocessor. On the other hand, there are trends towards more integrated and flexible architectures, where software components can be freely allocated to "heavy weight" computer units (Electronic Control Units; ECUs). One reason for this is that the number of ECUs is growing beyond control in a modern car (in the range of 100 in top of the line models). Letting SW from several suppliers, related to different sub-systems, execute on the same ECU has several benefits, including reduced number of ECUs, reduced cabling, reduced number of connection points (essential for system reliability), reduced weight, and reduced per-unit production cost. The downside is an increased risk of interference between the different sub-systems. Minimizing this risk and increasing efficiency and flexibility in the design process is the main motivation for SAVEComp and other efforts currently in progress (e.g. the EAST/EEA and AutoSar initiatives [1] [2]).

The safety-critical sub-systems we consider will in the foreseeable future have the following characteristics:

- Statically configured, i.e., the components used and their interconnections will essentially be decided at design or configuration time. Hence, the binding will be static, as opposed to the dynamic binding used in current component technologies.
- It will be essential to satisfy and provide proof of satisfaction of not only the functional behaviour, but also of timing and dependability quality attributes.
- The timing and dependability quality attributes will be strict, in the sense that they will be specified in terms of absolute bounds that must be satisfied.
- There will be additional, less critical, less static components executing on the same ECUs as the critical ones. The focus of SAVE is however not on these.
- The systems will be resource constrained, in the sense that the per-unit cost is a main optimization criterion, i.e., the use of computer and computing resources should be kept at a minimum.
- Due to the "product-line nature" of the industry, reuse of architectures, components and quality assessments should be supported.
- The contractual aspect of system and component models will in many cases be important as a tool for communication and ensuring quality in the integrator – supplier relation.

Looking more in detail at the timing quality attributes, SaveCCM should provide sufficient machinery to express and reason about the following types of timing attributes/requirements:

- End-to-end timing, i.e., it should be possible to determine (or guarantee) that the time from some event (e.g., sampling of a sensor value) to the time of some other event (e.g., providing a new control signal to an actuator) stays within specified bounds.
- Freshness of data, i.e., it should be possible to determine (or guarantee) that a datum has been generated no earlier than a specified bound before it is used by a specific component (e.g., that a sensor value has been sampled no earlier than 35ms before it is used by a specific component).

- Simultaneity, i.e., it should be possible to determine (or guarantee) that a set of data occur sufficiently close together in time (e.g., that the sampling of two sensors occur within 2ms).
- Jitter tolerances, i.e., it should be possible to determine (or guarantee) that the variation in latency between two events stay within specified bounds (e.g., that the variation in the time between subsequent (periodic) samplings of a sensor value stays within 2ms).

## 5. The SAVEComp Component Model

SaveCCM has its roots in previous models and design methods for embedded real-time systems, in particular Basement [5] and its extensions into the Rubus-methodology [3] [8], which (as mentioned above) is currently in industry use. SaveCCM, and its predecessors are designed specifically for the vehicular domain, which – in contrast with many of the current component technologies – implies that predictability and analysability are more important than flexibility. Hence, the model should be as restrictive as possible, while still allowing the intended applications to be conveniently designed. It is with this in mind we have designed SaveCCM.

### 5.1. Architectural elements

SaveCCM consists of the following main elements:

- *Components*, which are basic units of encapsulated behaviour, that executes according to the execution model presented below.
- *Switches*, which provide facilities to dynamically change the component interconnection structure (at configuration or run-time).
- *Assemblies*, which provide means to form aggregate components from sets of interconnected components and switches.
- *Run-time framework*, which provides a set of services, such as communication between components. Component execution and control of sensors and actuators.

Both switches and assemblies can be considered to be special types of components. Due to the difference in semantics we will, however, treat them as separate elements. Below, we will elaborate on these elements, their properties, and their attributes.

#### Functional interface

The functional interface of all architectural elements is defined in terms of a set of associated *ports*, which are points of interaction between the element and its external environment. We distinguish between input- and output ports, and there are two complementary aspects of ports: the data that can be transferred via the port and the triggering of component executions. SaveCCM distinguish between these two aspects, and allow three types of ports: (1) data-only ports, (2) triggering-only ports, and (3) data and triggering ports.

An architectural element emits trigger signals and data at its output ports, and receives trigger signals and data at its input ports. Systems are built from components by connecting input ports to output ports. Ports can only be connected if their types match, i.e. identical data types are transferred and the triggering coincides.

*Data-only ports* are one element buffers that can be read and written. Each write will overwrite the previous value stored. Output and input ports are distinct, in the sense that writing a datum to an output port does not mean that the datum is immediately available at the input port connected to the output port. This is to allow transfer of data between ports over a network or any other mechanism that does not guarantee atomicity of the transfer.

*Triggering-only ports* are used for controlling the activation of components. A component may have several triggering ports. The component is triggered when all input triggering ports are activated. Several output triggering ports may be connected to a single input triggering port, providing an “OR-semantics”, in the sense that the input port is triggered if at least one of its connected output ports is activated. Note that the input triggering port is active from the time of activation (triggering) to the start of execution of the component. Activations cannot be cancelled, and activating an active port has no effect.

*Data and triggering ports* combine data-only and triggering-only ports in the obvious way.

#### Execution model

Since predictability and analyzability are of primary concern for the considered application domain, the SaveCCM execution model is rather restrictive.

The basis is a control-flow (pipes and filter) paradigm in which executions are triggered by clocks or external events, and where components have finite, possibly variable, execution time.

On a high level, a component is either waiting to be activated (triggered) or executing. A component change state from waiting to executing when all input triggering ports are active.

In a first phase of its execution a component reads all its inputs. In its second execution phase the component performs all its computations based only on the inputs read and its internal state. In its third execution phase, the component generates outputs, after which it returns to its idle state waiting for a new triggering.

**External I/O**

Sensors and actuators (I/O) are accessed via enclosing components, in which the sensor/actuator values are part of the component's internal state.

**Timing**

Time is a first class citizen in SAVEComp. A global time base is assumed (a perfect clock). This perfect clock is accessed via special components, called triggers, which can trigger the activation of other components. To cater for the imperfection of real clocks, a triggering initiated at time t will arrive at the receiving component sometime in the interval t±δ .

**Switches**

As mentioned above, a switch provides means for conditional transfer of data and/or triggering between components. Switches allow configuration of assemblies. A switch contains a connection specification, which specifies a set of connection patterns, each defining a specific way of connecting the input and output ports of the switch. Logical expressions (guards; one for each pattern) based on the data available at some of the input ports of the switch are used to determine which connection pattern that is in effect.

It should be noted that a pattern does not have to provide connections for all ports, it is sufficient to only connect some input and some output ports.

Switches can be used for pre-run-time static configuration by statically binding fixed values to the data in some of the input ports, and then use partial evaluation to reduce the alternatives defined by the switch.

Switches can also be used for specifying modes and mode-switches, each mode corresponding to a specific static configuration. By changing the port values at run-time, a new configuration can be activated, thereby effectuating a mode-shift.

**Assemblies**

As mentioned above, component assemblies allow composite behaviours to be defined, and make it possible to form aggregate components from components and switches. In SaveCCM, assemblies are encapsulations of components and switches having an external functional interface, just as SaveCCM-components. Some of the ports of components and switches are associated/delegated to the external ports of the assembly.

Due to the strict (and restricted) execution semantics of SaveCCM components, an assembly does not satisfy the requirements of a component. Hence, assemblies should be viewed as a mechanism for naming a collection of components and hiding internal structure, rather than a mechanism for component composition.

**Quality attributes**

Handling of quality attributes, in particular those related to real-time and safety, is one of the main aspects of SaveCCM. A list of quality attributes and (possibly) their values is included in the specification of components and assemblies. In this paper we will only consider timing attributes. We will show how such attributes can be specified and used in analysis.

## 5.2. Specification and Composition Language

We will now outline the textual syntax used to define SaveCCM components and assemblies.

A SaveCCM system is an aggregate of component instances. A component instance is a named instance of a component type. A component type is either a basic component type or a component assembly type. A basic component type is defined as follows:

Components are specified by their interfaces, behavior and (quality) attributes. Interfaces are port-based and they specify input and output ports. Behavior identifies variables that express internal states, and actions that describe the component execution. Variables can be initiated by values from the input ports. Attributes describe different properties of the components. An attribute has a type, value and credibility (a measure of confidence of the expressed value). Credibility value, expressed in percentage is discussed in [11]. Ports include data or triggers or both. A simplified BNF specification of a component type is shown below. Actions are abstract specifications of the externally visible behavior of the component.

```
<component> ::= Component <typeName>
{<componentSpec>}
<componentSpec> :: =<Interface>  [<Behaviour>]
[<Attributes> ]
<Interface> ::= Inports: <port>[,<port>]+ ;
                Outports: <port>[,<port>]+ ;
<port> ::= <portName> : <portTypeName>;
<Behaviour> ::= Variables: <variables>+
Actions: <actions>+
<Variables> ::= <type> <name> [ = <value> | =
<port_name> ] ;
<actions> ::= { <action-program> }
<Attributes> ::= Attributes <attributeSpec>+ ;
```

```
<attributeSpec> ::=  <type> <name> = <value>
[:<credibility>]
<portType> ::= Port <Name> {<portSpec>};
<portSpec> ::= Data: <dataType|empty>;
               Trigger: <bolean> ;
```

Switches are specified as special types of components, however without actions and attributes. Depending on the switch state (condition) particular input and output ports are connected or disconnected.

```
<switch> ::= Switch <type> <name>{<swSpec>}
<swSpec> ::= <Interface>  <behaviour>
<Interface> ::= Inports: <port>[,<port>]+ ;
              Outports:  <port>[,<port>]+ ;
<port> ::= <portType> <portName> ;
<behaviour> ::= Switching: <cond>:<in-out-
connect> [,<in-out-connect>];
<in-out-connect> ::= <portName> -> <portName>
[,<portName> -> < portName>];
```

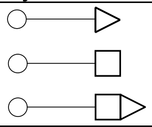
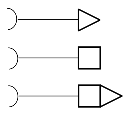
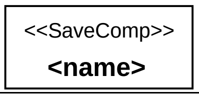
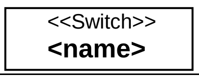
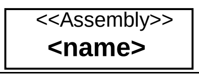
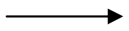
An assembly includes a set of components and switches that are “wired” together. Similar to components assemblies can be instantiated, which enables reusability on a higher level than the component level. However, the specification does not include a behaviour (variables and activities) part. Quality attributes are part of assemblies. The reason is that there are assembly properties which cannot be derived from the component properties but are applicable and can be measured on the assembly level.

```
<assembly> ::= Assembly  <assemblyType>
{<assemblySpec>}
<assemblySpec> ::= <Interface>  <Behaviour>
                   [<Attributes> ]
<Interface> ::= Inports: <port>[,<port>]+ ;
                Outports: <port>[,<port>]+ ;
<port> ::= <portType> <portName> ;
<Behaviour> ::= Components: <componentName>
[,<compomemtName >+] <connections>
<connections> ::=  Connections
<singleConnection> [,<singleConnection>]+
<singleConnection> ::= <portName> ->
<componentName.portName>
| <componentName.portName> -> <portName>
 |<componentName.portName> ->
<componentName.portName>
<Attributes> ::= Attributes <attributeSpec>+ ;
<attributeSpec> ::= <type> <name> = <value>
[:<credibility>];
```

In modelling and building systems we must create instances of these types and associate instances to tasks that execute on target systems. We will, however, in this paper not discuss these issues further, though our examples will contain some instantiations that we hope will be intuitive enough to be understood without further explanations.

## 5.3. Graphical Language

A subset of the UML2 component diagrams is adopted as graphical representation language. The interpretation of the symbols for provided and required interfaces, and ports are somewhat modified to fit the needs of SaveComp. The following symbols are used:

| Symbol | Interpretation |
|---|---|
|  | **Input ports** - The upper is an input port with a trigger, and no data. The middle symbol is an input port with data and no triggering, and the lower symbol is an input port with data and triggering. |
|  | **Output port** - Similar to the input ports, the upper is symbol is an output port with triggering functionality but with no data. The middle symbol is an output port with data but with no triggering, and the lower symbols indicates an output port with both data and triggering. |
|  | **Component** - A component with the stereotype changed to SaveComp corresponds to a SaveCCM component. |
|  | **Switch** - components with the stereotype switch, corresponds to switches in SaveCCM. |
|  | **Assembly** - components with the stereotype Assembly, corresponds to assemblies in SaveCCM. |
|  | **Delegation** - A delegation is a direct connection from an input to -input or output to -output port, used within assemblies. |

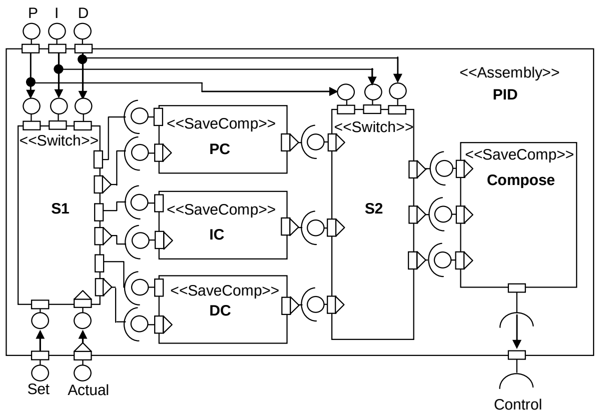
## 5.4. Simple examples

We will give a few examples to illustrate SaveCCM. In the examples we will use our graphical language, and for selected architectural elements also the textual format.

### Static configuration

By static configuration we assume instantiation of assemblies and the included components. For example we specify a general controller, which can be configured to be a P, I, D, PI, PD, ID, or PID controller. Switches are used to express this.

Graphically we can illustrate PID as follows:



The following is the same example expressed in the specification and composition language:

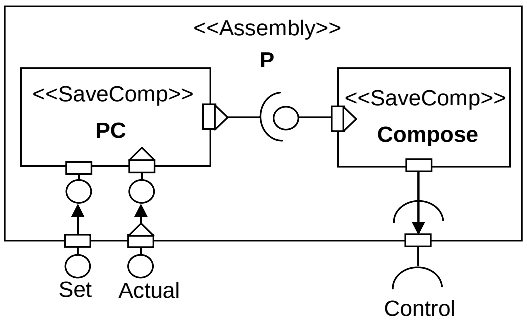
```
Assembly PID  {
   Inports: P:Pport, I:Iport, D:Dport,
       Set:Setport, Actual:Actualport;
   Outports: Control:Controlport;
   Components: PC:PCtype, IC:ICtype,
     DC:DCtype, Compose :Ctype, S1:S, S2:Z;

  PortConnect:
   P->{S1.P,S2.P}, I->{S1.I,S2.}, D->
   {S1.D,S2.D},Set->S1.setin, Actual->
   S1.actualin,S1.actualoutp->P.actual,
   S1.actualouti-> I.actual, S1.actualoutd->
   D.actual,S1.setoutp-> P.set, S1.setouti->
   I.set, S1.setoutd->D.set, P.control->S2.p,
   I.control->S2i, D.control-> S2.d, S2.pp->
   Compose.p, S2.ii->Compose.i, S2.dd->
   Compose.d, Compose.control->control
}
Switch S {
   Inports: P:Pport, I:Iport, D:Dport,
       setin:Setport, actualin:Actualport;
   Outports: actualp:Actualport,
     actuali:Actualport, actuald:Actualport,
     setoutp:Setport, setouti:Setport,
     setoutd:Setport
   Switching:
      P: setin->setoutp, actualin->actualp;
      I: setin->setouti, actualin->actuali;
      D: setin->setoutd, actualin->actuald;
}
Switch Z {
   Inports: P:Pport, I:Iport, D:Dport,
      p:Setport, i:Setport, d:Setport;
   Outports: pp:Setport, ii:Setort,
       dd:Setport;
   Switching:
      P: p->pp; I: i->ii; D: d->d;
}
```

Like components, assemblies can be reused. When creating a component instance or an assembly we can statically bind port values to constants. For instance if the component type PID is instantiated with P set to true, and I and D set to false, we will (by partial evaluation) obtain the following component. This configuration is supposed to be done automatically by a configuration tools.

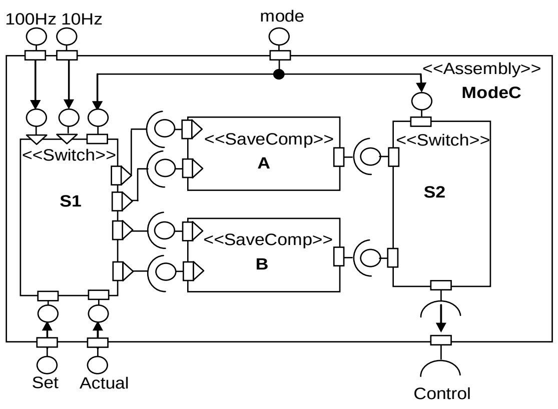
```
Assembly P:PID (P.val=true, I.val=false,
D.val=false) {
   Inports: Set:Setport, Actual:Actualport;
   Outports: Control:Controlport:
   Components: PC:PCtype, Compose:Ctype
   PortConnect: Set->P.set, Actual->P.actual
      P.control-> Compose.p, Compose.control->
      control;
}
```

The graphical interpretation is shown below.



### Mode shift

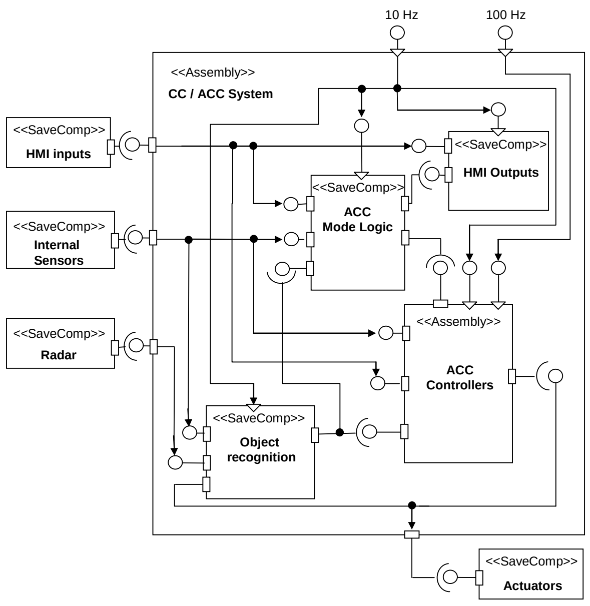
We specify a component (ModeC) with two externally determined modes: idle and busy. In mode idle control algorithm A should run at 10Hz and in mode busy control algorithm B should run at 100Hz. Graphically we illustrate ModeC as follows:



## 6. The Cruise Control Example

To further illustrate the use of SaveCCM we demonstrate a simple design of an Adaptive Cruise Control system (ACC), as an example of an advanced function in a vehicle. An ACC system helps the driver to keep the distance to a vehicle in-front, i.e., it autonomously adapt the velocity of the vehicle to the velocity and distance of the vehicle in front.

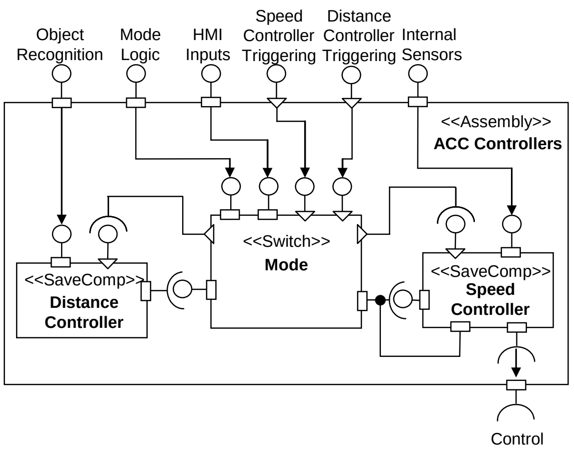
The following figure visualises a suggested ACC system using SaveCMM:



The above system can be divided into three major parts: input, control, and actuate. Our focus will be on the control part that is encapsulated in the CC/ACC System assembly. The CC/ACC system consists of three components and a switch:

- **Object recognition** is a component that has responsibility to determine if there is a vehicle in front and in that case estimate the distance and relative velocity. It is triggered by the CC/ACC 10 Hz triggering port, and has a Worst Case Execution Time (WCET) of 30 ms.
- **ACC controllers** is an assembly implementing two cascaded controllers. The inner controller is for speed control and can be used for normal Cruise Control (CC), while the outer handles distance control. The assembly has two triggering ports, one for the inner loop, and one for the outer.
- **HMI outputs** is a component that gives information to the driver through the vehicle computer display, e.g., information about the vehicle state and latest request. The component is triggered by the CC/ACC systems triggering port bound to 10 Hz. The WCET is 2 ms.
- **ACC mode logic** is a component implementing the logic for shifting modes depending on the state of the vehicle, inputs by the driver and from the environment (vehicles in front). The different modes are CC, ACC, and standby. It is triggered by the 10 Hz port. The WCET is less than 1 ms.

A diagram showing the internal design of the assembly ACC Controllers is provided in the following figure:



In the figure the name of the component attached to in-ports is written above each port. A brief presentation of the different components in the assembly is given below.

- **Distance Controller** is a pure controller component implementing a control algorithm; it handles distance control and is the component in the outer loop. The WCET is 20 ms, and it is triggered at 10 Hz.
- **Mode** is a switch, which depending on the actual mode of the controller activates and deactivates the both controller components. The switch also switches the input of the speed controller, between HMI Inputs (CC functionality) and from the control signal of the outer loop controller (ACC functionality).
- The **speed controller** executes with a rate five times faster than the rate of the distance controller due to faster dynamics, it control the speed of the vehicle. The WCET is 5 ms.

As illustrated by the example, SaveCCM is designed to seamless and easily support typical requirements that arise when designing advanced vehicular functionality, e.g., connections with data, triggering and both, assemblies, feedback, and mode changes.

As an illustration how the above SaveCCM specification can be used in analysis of timing properties, let us (somewhat simplified) assume that the CC/ACC System will be exclusively allocated to an ECU and that each component is allocated to a single task. We further assume that the tasks are executing under a fixed priority (FPS) real-time kernel, with a zero execution time overhead, and that the deadline attributes of the components are defined to be equal to the periods. Given this, and using deadline monotonic priority assignment, together with the execution time attributes of the components, we can derive the following task set for the ACC mode:

| **Task** | **Period (ms)** | **WCET (ms)** | **Prio** |
|---|---|---|---|
| Object Recognition | 100 | 30 | 5 |
| Mode Logic | 100 | 1 | 4 |
| HMI Outputs | 100 | 2 | 3 |
| Distance Controller | 100 | 20 | 2 |
| Speed Controller | 20 | 5 | 1 |

The task set can be used as input to standard fixed-priority schedulability analysis tools (e.g. [7]). We can use such a tool to verify if the deadline attributes are satisfied. By applying this analysis we find that the all deadline attributes are satisfied, hence we can from now on treat these attributes as properties of the current configuration of the CC/ACC System.

## 7. Conclusions and further work

We have presented SaveCCM, a component mode intended for embedded control applications in vehicular systems. In contrast with most current component technologies, SaveCCM is sacrificing flexibility to facilitate analysis; in particular analysis of dependability and real-time. We illustrate SaveCCM by a simple example, where we also, as an example of timing analysis, show that SaveCCM models are amenable to schedulabilty analysis.

This paper covers only parts of the component specifications. In our future work we will provide a complete and formal definition of SaveCCM, as well as linking it to further methods and tools for both dependability and timing analysis. Parts of the specifications not discussed here include actions and attributes describing dynamic behaviour of the components and attribute values that are used for reasoning about system properties. Furthermore, we will in association with our industrial partners evaluate SaveCCM in "real-life" case-studies.

## 8. References

[1] EAST, Embedded Electronic Architecture Project, http://www.east-eea.net/

[2] Autosar homepage: http://www.autosar.org/

[3] D. Isovic, C. Norström, Components in Real-time systems, Chap. 13 in I. Crnkovic, M. Larsson, Building Reliable Component-Based Software Systems, 2002, ISBN 1-58053-327-2

[4] Merijn de Jonge, Johan Muskens and Michel Chaudron Scenario-Based Prediction of Run-time Resource Consumption in Component-Based Software Systems, CBSE6 proceedings, http://www.csse.monash.edu.au/~hws/cgi-bin/CBSE6/Proceedings/proceedings.cgi

[5] Hans Hansson, H. Lawson, O. Bridal, Christer Norström, S. Larsson, H. Lönn, M. Strömberg, BASEMENT: An Architecture and Methodology for Distributed Automotive Real-Time Systems, IEEE T. on Computers, 46(9):1016-1027, Sept. 1997.

[6] Peter Müller, Christian Stich, Christian Zeidler, Components @ Work: Component Technology for Embedded Systems, 27th International Workshop on Component-Based Software Engineering, EUROMICRO 2001

[7] MAST - Modeling and Analysis Suite for Real-Time Applications, http://mast.unican.es/

[8] C. Norström, M. Gustafsson, K. Sandström, J. Mäki-Turja, N.-E. Bånkestad, "Experiences from Introducing State-of-the art Real-time Techniques in the Automotive Industry", Proceedings 8th IEEE Int'l Conf. on Engineering of Computer-based Systems. IEEE 2001.

[9] R. van Ommering, F. van der Linden, and J. Kramer; The Koala component model for consumer electronics software. IEEE Computer, 33(3):78–85, March, 2000

[10] SAVE Project, http://www.mrtc.mdh.se/SAVE/

[11] M. Show,"Thruth vs Knowledge: The difference between what a component does and what we know it does", Proc. 8th Intl-workshop on software specification and design, IEEE 1996

[12] Wallnau, Kurt C. & Ivers, James. Snapshot of CCL: A Language for Predictable Assembly, Technical report CMU/SEI-2003-TN-025

[13] Wallnau, Kurt C. Volume III: A Technology for Predictable Assembly from Certifiable Components, Technical report CMU/SEI-2003-TR-009.