The SaveCCM Language Reference Manual

John Håkansson

Dept. of Information Technology
Uppsala University, Uppsala, Sweden

Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson,
Mikael Nolin, Thomas Nolte, Paul Pettersson

Dept. of Computer Science and Electronics
Mälardalen University, Västerås, Sweden

Contents

Chapter 1

Introduction

This language reference describes the syntax and semantics of SaveCCM, a component modeling language for embedded systems designed with vehicle applications and safety concerns in focus. The SaveCCM component model was defined within the SAVE[1] project. The SAVE components are influenced mainly by the Rubus component technology [8], with a switch concept similar to that in Koala [10]. The semantics is defined by a transformation into timed automata with tasks, a formalism that explicitly models timing and real-time task scheduling [6].

The purpose of this document is to describe a semantics of the SAVE component modeling language, which can be used to describe timing and functional behavior of components. The model of a system is in some cases an over approximation of the actual system behavior. An implementation of a model can resolve non-determinism e.g. by merging tasks or assigning a scheduling policy (such as static scheduling or fixed priority, preemptive or not) that will resolve the non-determinism.

In [1] a component technology called SaveCCT is presented, which uses SaveCCMas component modeling language. In addition to modeling and specification of components, SaveCCT supports synthesis of code for the run-time systems, and analysis of models using e.g., the model-checking tool TIMES [3]. Compositional reasoning of SaveCCMfor safety analysis is discussed in [5, 4].

The rest of this document is organized as follows: in Chapter 2 we describe the syntax of SaveCCM, and give informal semantics. In Chapter 3 we define a core language, which is a simpler language used to define the exact meaning of SaveCCM constructs. Chapter 4 gives formal semantics of the core language, and Chapter 5 defines SaveCCM in terms of the core language.

[1]SAVE is a project supported by Swedish Foundation for Strategic Research. See `http://www.mrtc.mdh.se/SAVE/` for more information.

Chapter 2

The SaveComp Component Model

The SaveComp Component Model (SaveCCM) formalises the component concept of the SaveComp component technology, and defines how components can be combined to create systems [7]. For use in the vehicular systems domain, the component model should support the development of resource-efficient systems and thus the run-time framework governing e.g., component communication, must be lightweight. Another requirement is that system behaviour should be predictable, both functionally and with respect to timeliness and resource usage.

SaveCCM is based on a textual XML syntax, and a somewhat modified subset of UML2 component diagrams is used as a graphical notation. The semantics is formally defined by a two-step transformation, first from the full language to a similar but simpler language called SaveCCM Core, and then into timed automata with tasks. In this chapter we will use the graphical notation only (the XML schema is described in Appendix A), and present the semantics informally. The graphical notation is presented in Figure 2.1.

In SaveCCM, systems are built from interconnected elements with well-defined interfaces consisting of input- and output ports. The three element categories; components, switches and assemblies, are described in more detail below. The model is based on the control flow (pipes-and-filters) paradigm, and an important feature is the distinction between data transfer and control flow. The former is captured by connections between *data ports* where data of a given type can be written and read, and the latter by *trigger ports* that control the activation of components. A port can also have both triggering and data functionality.

This separation of data and control flow results in a flexible model that supports both periodic and event-driven activities, since on a system level, execution can be initiated by either clocks or external events. It also allows components to exchange data without handing over the control, which simplifies the construction of, e.g., feedback loops and communication between sub-systems running

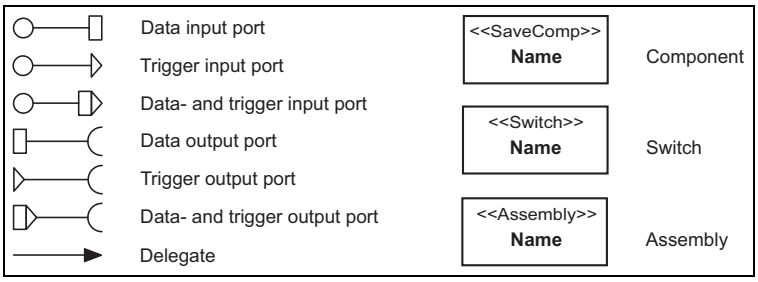

Figure 2.1: The graphical notation of SaveCCM.

at different frequencies.

Another aspect of explicit control flow is that the resulting design is sufficiently analysable with respect to temporal behaviour to allow analysis of schedulability, response time, etc., factors which are crucial to the correctness of real-time systems.

2.1 Components

Components are the main architectural element in SaveCCM. In addition to input and output ports, the interface of a component contains a series of quality attributes, each associated with a value and possibly a confidence measure. These attributes could include, for example, (worst case) execution time information for a number of target hardware configurations, reliability estimates, safety models, etc. The quality attributes are used for analysis, model extraction and for synthesis.

The concrete functionality of a component is typically provided by a single entry function implemented in C, but the model also allows the use of more complex components that consist of a number of possibly communicating tasks. In both cases, no intercomponent dependencies are permitted, except those explicitly captured by the ports.

A component is initially inactive. It remains in this state until all input trigger ports have been activated, at which point it switches to the executing state. We say that the component has been *triggered*. The component execution starts with a *read* phase, where the current value at each input data port is stored internally to ensure consistent computation. The component then performs the associated computations on the basis of this input and possibly an internal state. When the *execute* phase is over, i.e., when the function has been computed or, in the case of a more complex component, when all tasks have finished, the *write* phase writes output to the output data ports. Finally, the input trigger ports are reset and all outgoing trigger ports are activated, after which the component returns to the idle state.

This strict "read-execute-write" semantics ensures that once a component is

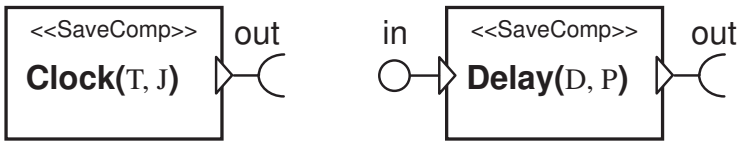

Figure 2.2: Clock and Delay components.

triggered, the execution is functionally independent of any concurrent activity. In particular, a component produces the same output with preemptive and non-preemptive scheduling, i.e., whether or not a task may be interrupted by another task during its execution. The "read-execute-write" semantics also facilitates analysis, since component execution can be abstracted by a single transfer function from input values and internal state to output values.

In order to manipulate timing of triggers we introduce two special component types, *Clock* and *Delay*. Their graphical syntax is shown in Figure 2.2. A *Clock* is a trigger generator, with parameters T and J for period and jitter, respectively. A new period starts every T time units, and a clock generates a trigger within J time units after the start of each period. A *Delay* component, with parameters D and P for delay and precision, will delay within D and $D+P$ time units from receiving a trigger until generating a trigger.

2.2 Composite Components

A composite component is a special case of a component, where the behaviour is specified by an internal composition. We introduce the notation examplified in Figure 2.3 to show the internal composition of a composite component. This expanded notation may be collapsed into the standard SaveCCM component notation.

The grey area illustrates the separation between the internal composition and the externals of the component. The dashed lines show how data is transferred from input ports in the read phase, or to output ports in the write phase. Note that triggering is not transferred in this way. Instead, all trigger ports to the internal composition will become active when the composite component becomes active. All triggering from the internal composition is discarded, since the composition is required to activate its output triggers when it becomes passive.

A composite component becomes active when all its input trigger ports become active (i.e. when it is triggered). In the read phase, data is transferred and internal components activated. The execute phase will perform computations of internal components, until no internal component is triggered or active. In the write phase data is transferred to external components, and the composite component becomes inactive (idle).

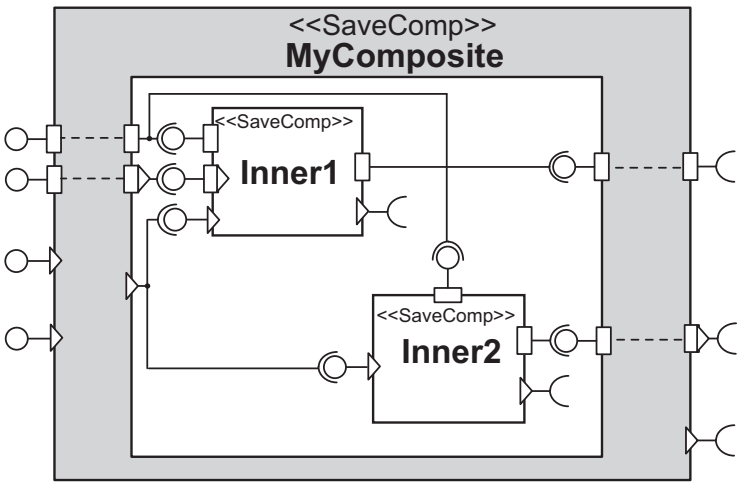

Figure 2.3: An expanded view of a composite component (by example).

2.3 Switches

The switch construct in SaveCCM is similar to that in Koala [10]. Switches provide the means to change the component interconnection structure, either statically for pre-runtime static configuration, or dynamically, e.g., to implement modes and mode switches. The switch specifies a number of connection patterns, i.e., partial mappings from input to output ports. Each connection pattern is guarded by a logical expression over the data available at the input ports of the switch, defining the condition under which that pattern is active.

If fixed values are supplied to ports used in connection pattern guards, partial evaluation can determine that parts of a switch will remain unchanged during runtime. Such static parts are optimised into ordinary connections, and components that are rendered unreachable as a consequence are omitted in the final system.

It should be noted that switches, in contrast with components, are not triggered. Instead, they respond directly to the arrival of data or a trigger signal at an input port and immediately relay it according to the currently active connection patterns. Switches perform no computation other than the evaluation of connection pattern guards.

2.4 Assemblies

Assemblies are encapsulated sub-systems. The internal components and interconnections are hidden from the rest of the system, and can be accessed only indirectly through the ports of the assembly. Like switches, assemblies are not triggered. Data and trigger signals arriving at a port are immediately relayed

to the outgoing connections.

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

2.5 Ports

As mentioned above, we distinguish between input and output ports, and between trigger ports and typed data ports. Component input ports, the output ports of the whole system, and switch input ports that occur in some connection pattern guard, are one-place buffers with overwrite semantics. The remaining ports, i.e. component output ports, assembly ports and switch ports that do not occur in any guard, are just conceptual interaction points through which data passes immediately.

All ports are named, all data ports are typed and may have an initial value. Names, types and values may be hidden in the graphical notation, but when shown appear as a label next to the port. The format of the label is name : type = value, name : type or just name. When the type of a data port is hidden its initial value is also hidden.

An *external* port is a port that is not connected with any other port, but has an extra label mapping it to some external entity. Example of external entities that can be mapped this way are I/O-ports, interrupts, and real-time database pointers [9]. The format of the label depends on the external entity to which it is mapped. Example labels are $inport(0x080f)$ for an input register at address 80f (hex), or $db\_ptr(q)$ for a database pointer, initialized by a query q (this query should be formulated in such a way that the result is always a single data element). External ports are not allowed internally within a composite component. Visualisation using entity type specific icons is encouraged, replacing the circle (for an input port) or semi-circle (for an output port).

A trigger output port can only be connected to trigger input ports, a data output port can only be connected to data input ports of compatible type, and a combined port can be connected to trigger, data or combined input ports of compatible type.

2.6 Connections

There are two types of connections: immediate and complex. *Immediate connections* represent loss-less, atomic migration of data or trigger signals from one port to another, as would typically be the case between components located on the same physical node. For distributed systems, and in particular during early design stages before the deployment of components to nodes has been determined, a more flexible connection concept is convenient. This is provided by *complex connections* that represent data and control transfer over channels with

possible delay or information loss. The detailed characteristics of a particular complex connection are explicitly modelled by a timed automaton to capture, e.g., delay constraints, buffer sizes, or the possibility of faults.

As in UML2, a connection from an assembly input port to an input port of an internal element, or from an internal output port to an assembly output port, is denoted by a delegation arrow, but semantically they are the same as ordinary connections from output to input ports.

Chapter 3

SaveCCM Core Syntax

We define a minimal component language, SaveCCM Core, from which we can derive the constructs of the SaveComp component model. This simplifies the definition of semantics, and makes it more flexible as new constructs can easily be derived. The core syntax consists of three modelling elements: basic components, composite components, and conditional connections. Using these we can describe all constructs in the SaveCCM language.

Each modelling element has a set of ports, through which it can interact. Each port is either an input port or an output port, as well as either a data port or a trigger port. A data port has a type associated with it. An input data port of a component is associated with a variable of the same type as the port holding the latest value written to the port. An input trigger port is associated with a boolean variable determining if the trigger port is *active*.

Common for basic components and composite components is that they have exactly one external output trigger port. For a component C we will write $\mathit{trigger\_out}(C)$ when referring to this port.

3.1 Basic Component

An example of the graphical syntax for basic components is shown in Fig. 3.1 (a). The component C_1 has three input ports and two output ports. Trigger ports are annotated with a small triangle, as for example port p_3. When the port p_3 becomes active the component is triggered, since p_3 is the only input trigger port. For the component C_1 we have the output trigger port $\mathit{trigger\_out}(C_1) = p_5$. In addition to its ports a component is characterized by its behaviour, describing the internal computation of the component.

We will model the internal behaviour of a basic component using a timed automaton with tasks [3]. For a simple component this could be a single task released when the component is triggered. A more complex component can have several tasks, possibly with intricate dependencies between them. The automaton has a special *exit* location with no outgoing edges. When this location is

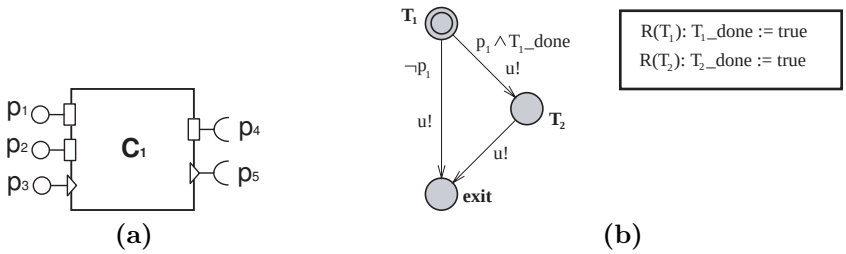

Figure 3.1: **(a)** A basic component C_1 with three input ports and two output ports. **(b)** Timed automaton with tasks, describing the behaviour of component C_1.

reached, and all released task instances have finished executing, the component becomes *idle* again. Locations can be labelled with tasks, and when such a location is reached the corresponding task is released for scheduling. Each task T_i is associated with a computation time $C(T_i)$, a deadline $D(T_i)$, and a sequence of assignments $R(T_i)$. The assignment $R(T_i)$ will update data variables when the task computation has completed. We will write *behaviour*(C) when referring to the automata modelling the internal behaviour of a component C.

The automaton in Fig. 3.1 (b) describes the behaviour of the component C_1. Two of the locations are labelled with tasks T_1 and T_2, the third is the exit location. In our example, the task T_2 depends on data computed by T_1. The task assignments $R(T_1)$ and $R(T_2)$ update the variables T_1_done and T_2_done so they can be used to test for task completion. The input data port p_1 is used to determine if task T_2 should be executed. The type of port p_1 is boolean. When the component is triggered, the task T_1 is released. The assignment $R(T_1)$ updates the variable T_1_done to true when task T_1 completes. If the value at port p_1 is true the task T_2 is released after T_1 completes, and before the *exit* location is reached.

3.2 Composite Component

A composite component is a component with its internal behaviour defined by a composition of internal components. The component C_4 seen in Fig. 3.2 has seven external ports p_1 through p_7, and five internal ports p'_1 through p'_5. When the trigger ports p_3 and p_4 become active, C_4 is triggered and becomes executing.

The connections between external and internal ports is provided by a component framework, to enforce a behaviour similar to that of a basic component. The contents of external input data ports are copied to internal output data ports when the composite component is triggered, and internal input data ports are copied to external output data ports when the composite component becomes idle again. There is a single internal output trigger port, which becomes

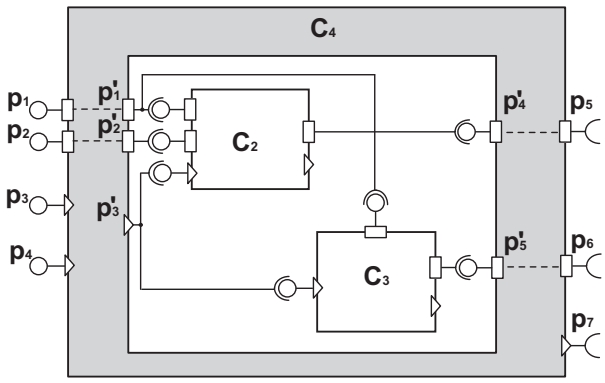

Figure 3.2: A composite component composed of two internal components. The dashed lines illustrate that the internal components are not directly connected to the external ports of the composite component.

active when the composite component is triggered. The external output trigger port becomes active when the composite component becomes idle again.

A composite component consists of external ports, internal ports, internal connections and internal components. For each external data port, there is a corresponding internal data port of the same type. For a composite component C we will write *trigger_in*(C) and *trigger_out*(C) when referring to the unique internal and external trigger output port, respectively.

3.3 Conditional Connection

The conditional connection is a connection with an activating condition, introduced to enable dynamic configuration of a model in such a way that it will become a static configuration when its parameters are fixed.

The graphical syntax of conditional connections is shown i Fig. 3.3, where (a) connects data ports and (b) connects trigger ports. It is a connection from port p_1 to port p_2 that is active when the expression $p_3 \wedge p_4$ holds. The ports p_3 and p_4 are the *setports* of the connection, containing data used in the expression. The setports of a conditional connection are not trigger ports. The connections in Fig. 3.2 have no conditions, and are drawn as lines. The lines are special cases of conditional connections, with no setports and a condition that is always `true`.

For a conditional connection x, *from*(x) is the sending port, *to*(x) is the receiving port, *setports*(x) are the setports of the connection and *expr*(x) is a boolean expression over the setports. The ports *from*(x) and *to*(x) must be of the same type.

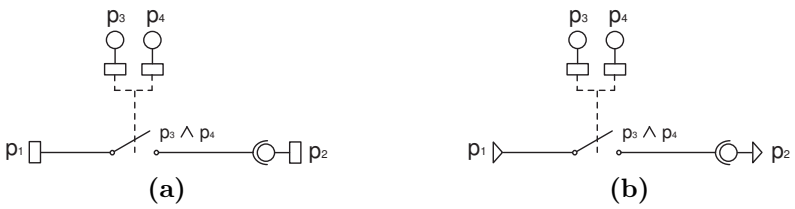

Figure 3.3: A conditional connection with two setports, the connection is active when both setports are true. **(a)** connects two data ports, **(b)** connects two trigger ports.

Chapter 4

SaveCCM Core Semantics

We define the semantics of SaveCCM Core by describing a translation to networks of timed automata [2] extended with tasks [3]. We extend this further with *operations*. An operation is a sequence of statements, such as variable updates or conditional **if**-statements. As mentioned above, locations can be labelled with tasks. When such a location is reached the corresponding task is released for scheduling.

In order to model a transition which is taken as soon as its guard becomes satisfied, we introduce an urgent channel u which is always available for synchronization. For a component C we introduce the variable $idle_C$, and for its ports p variables ext_p, int_p and $active_p$. For a conditional connection we introduce ext_p for its setports.

The variable ext_p represent the observable data value at an input data port or setport. The boolean variable $active_p$ is `true` when the input trigger port p has been activated. Basic components use int_p to keep an internal working copy of port data. The boolean variable $idle_C$ is `true` when component C is idle, and false otherwise. It is used for composite components to determine when all its internal components are idle.

Each component in a SaveCCM system is modelled as a separate timed automaton, and the system is modelled as the parallel composition of these automata.

4.1 Basic Component

The full SaveCCM language imposes some restrictions on the component behaviour that should be addressed in the core language as well. For example, the so-called *read–execute–write* semantics specifies that input ports may only be accessed at the very start of each invocation, and output ports are only written to at the end.

The automaton $behaviour(C)$ describes the response of a component being triggered. To define its reactive behaviour we augment this automaton with a

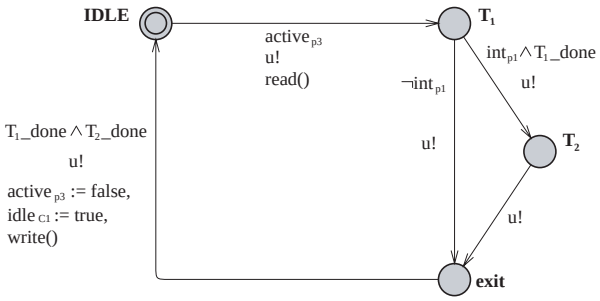

Figure 4.1: Semantics of component C_1 in Fig. 3.1.

location *idle* and two edges, one from idle to the initial location of *behaviour*(C), and one from the *exit* location of *behaviour*(C) to *idle*. We also replace all port references p with references to the corresponding internal variable int_p.

A component remains in *idle* until all its input trigger ports are active. On the transition from idle, internal port variables are updated from the corresponding input ports. When the *exit* location is reached, and all released task instances have finished executing, the component becomes idle again. On the transition from exit to idle, input trigger ports are deactivated, and output ports are forwarded by the component framework.

Fig. 4.1 shows the semantics of the component C_1 in Fig. 3.1. When the port p_3 becomes active, the component is triggered and the urgent transition from idle is enabled. The *read*() operation invoked by this transition updates the internal port variables int_{p_1} and int_{p_2} from external port variables ext_{p_1} and ext_{p_2}, respectively. The variable int_{p_1} is used in a guard to determine if task T_2 should be released after T_1 has completed. The transition from *exit* to *idle* is enabled when the tasks T_1 and T_2 have completed. The transition will deactivate port p_3, set $idle_{C_1}$ to `true`, and invoke the *write*() operation.

The *write*() operation is considered a part of the component framework. It is invoked by the internals of a component, and implements the behaviour of external connections. The operation is a sequence of invocations $write_x()$ for each connection x from an output of the component, as described in Section 4.3. The order in which the $write_x()$ operations are invoked can effect which connections are active, since one connection can update a setport of another. Therefore, we introduce a dependency relation between connections c_1 and c_2 leading from the same component,

$$before(c_1, c_2) \quad \text{iff} \quad to(c_1) \in setports(c_2)$$

and require that the $write_x$ operations are ordered in accordance with these dependencies. For cyclic dependencies, any ordering is considered correct.

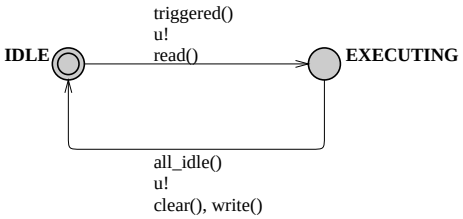

Figure 4.2: Semantics of a composite component.

4.2 Composite Component

The role of this construct is to enforce that the combined behaviour of the internal components conforms to the component semantics imposed by SaveCCM. In particular, the component as a whole should be triggered when all input trigger ports are active, and the input and output ports are only available at the start and end of execution, respectively.

The automaton in Fig. 4.2 describe the semantics of composite components. The guard *triggered*() enables the transition from *idle* when all input trigger ports are active. Data is transferred to internal ports by *read*(), which also activates the internal output trigger port $trigger\_in(C)$ of the composite component C. As internal components are triggered, they start executing. The guard *all_idle*() enables the transition back to *idle* when $idle_{C'}$ is true for all internal components C'. Input trigger ports are deactivated by *clear*(), which also updates $idle_C$ to true for the composite component C. The *write*() operation works similarly to that of a basic component.

For the component C_1 in Fig. 2.3, *triggered*() holds when both p_3 and p_4 are active. The *read*() operation performs $write_x()$ operations to update the input ports of the internal components C_2 and C_3, which also updates $idle_{C_2}$ and $idle_{C_3}$ to false by the trigger connections. When $idle_{C_2}$ and $idle_{C_3}$ become true, *all_idle*() holds and C_1 becomes idle. On the transition to *idle*, p_3 and p_4 are deactivated by *clear*(), which also updates $idle_{C_1}$ to true. The *write*() operation forwards values at ports p'_5 and p'_6 in a sequence of $write_x()$ operations for connections x from ports p_5 and p_6.

4.3 Conditional Connection

The semantics of a conditional connection x is described by a $write_x()$ operation. The operation will update the input port $to(x)$ from an output port $from(x)$ only if $expr(x)$ holds. For a data connection, the external port variable of $to(x)$ is updated with the internal port variable of $from(x)$. For a trigger connection, the port $to(x)$ is activated and if all input trigger ports of a component C become active the variable $idle_C$ is updated to false.

if $ext_{p_3} \wedge ext_{p_4}$ **then**
$ext_{p_2} := int_{p_1}$
end if

(a)

if $ext_{p_3} \wedge ext_{p_4}$ **then**
$active_{p_2} :=$ true
if $active_{p_5}$ **then** $idle_C :=$ false
end if

(b)

Figure 4.3: The $write_x$ operation for the conditional connections in Fig. 3.3 (a) and (b).

For the conditional connection in Fig. 3.3, where p_2 and p_5 are the input trigger ports of a component C, we define $write_x()$ as in Fig. 4.3. If the condition $p_3 \wedge p_4$ holds, port p_2 is updated from port p_1. For the data connection in (a), the external port variable of the input port p_2 is updated from the internal port data of the output port p_1. For the trigger connection in (b), the input trigger port p_2 is activated. If port p_5 is also active the component C is no longer idle.

Chapter 5

SaveCCM Semantics

The SaveCCM modelling language is built around the same concepts of ports, components and connections as the core language, but there are some differences. SaveCCM components can have any number of output trigger ports, and there is a port type that combines data and triggering. The full language also contains *assembly* and *switch* constructs, which are not in the core language. The constructs of SaveCCM are described below, and we show how they can be expressed in the core language.

5.1 SaveCCM Constructs

The PI controller depicted in Fig. 5.1 will be used as an example when describing the syntax and semantics of SaveCCM constructs. PID controllers are common for continuous control of for example fuel injection in vehicles. We have restricted the example to PI control to reduce the level of detail in the example.

5.1.1 Connections

As in the core language, connections define how data and control can be transferred between components, but SaveCCM connections have a very weak semantics compared to the connections in the core language. In general, nothing is said about the time it takes to migrate data over a connection, if data can be lost in the process, the order in which it arrives, etc. This loose concept of connection is useful in early stages of system design, e.g., before deploying components to the different nodes of a distributed system. For detailed analysis of the system, quality attributes such as maximum delay can be provided. In order to define a detailed semantics for connections that are specified in detail, while still allowing loosely specified connections, we categorise connections as either *immediate* or *complex*. The former represent loss-less, atomic migration of data or triggering from one port to another, as would typically be the case

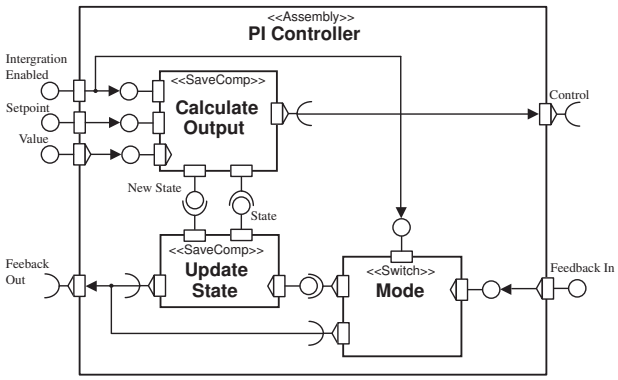

Figure 5.1: An example assembly for a PI controller.

between components residing on the same node. Any other type of connection is categorised as complex. Immediate connections have direct formal semantics, whereas complex connections are handled indirectly by explicit modelling of the connection behaviour.

5.1.2 Switches

In addition to basic and composite components, there are two more component types in the full SaveCCM language. Switches are lightweight components used to change the component interconnection structure, either statically for pre-runtime static configuration, or dynamically, e.g., to implement modes and mode switches. The switch specifies a number of connection patterns, i.e., partial mappings from input to output ports. Each connection pattern is guarded by a logical expression over the data available at the input ports of the switch, defining the condition under which that pattern is used. Switches perform no computation other than the evaluation of connection pattern guards.

The switch Mode in the PI controller has two configurations, depending on the boolean value of the setport Integration Enabled. When the setport is true the port Feedback In is connected to Update State, otherwise Feedback In is connected to Feedback Out. The purpose of Mode is to bypass the Update State component when integration is disabled.

5.1.3 Assemblies

Assemblies are encapsulated subsystems, just like composite components. The internal interconnections and components are hidden from the rest of the system, and can be accessed only through the ports of the assembly. They differ from compositions in that they provide syntactic abstraction only, meaning that an assembly does not necessarily behave like a basic component.

The PI controller is an example of how an assembly can violate the *read–execute–write* semantics that is expected from basic components and compositions. This is because in a cascaded control loop, constructed as a chain of PI controllers, several Calculate Output instances will compute the control signal, and after the actuator has been updated the Update State instances will compute the next control state. The two trigger ports trigger separate parts of the PI controller, and control is passed on differently afterwards.

If, instead, the PI controller was designed as a composite component, it would remain idle until triggered by both Value and Feedback In. Then, the internal components would be invoked, and once both had finished, data and control would be passed on to both Control and Feedback out.

5.2 Translating SaveCCM into SaveCCM Core

Basic components and compositions have direct core language counterparts. The differences regarding output trigger ports and ports with combined data and triggering, are handled as part of the connection translation described below. A basic SaveCCM component corresponds to a basic core component with a behaviour automaton that captures the behaviour of the associated code. Each composite component results in a corresponding composite core component, with the same (but transformed) contents. Assemblies and switches are not represented directly by any core construct, but they influence the translation of connections.

5.2.1 Clock

A *Clock* component (Figure 2.2) is a trigger generator, with parameters T and J for period and jitter, respectively. A new period starts every T time units, and a clock generates a trigger within J time units after the start of each period. All clocks are independent, meaning that there is no assumption that clocks are started simultaneously.

The behaviour automaton of a core component corresponding to *Clock(T,J)* is shown in Figure 5.2 (a). The automaton uses a clock x and a boolean variable *first*. Initially, x is zero and *first* is true. Since a *Clock* has no input ports, it is immediately activated (all its input trigger ports are active). The first time the clock becomes active, it waits between zero and $T + J$ time units before the exit location is reached. This initiates the write phase, where the output trigger port is activated. Any other time the clock waits between T and $T + J$, resetting x after exactly T time units. When the component becomes inactive it is immediately activated again, since it has no input trigger ports.

5.2.2 Delay

A *Delay* component (Figure 2.2), with parameters D and P for delay and precision, will delay within D and $D + P$ time units from receiving a trigger until

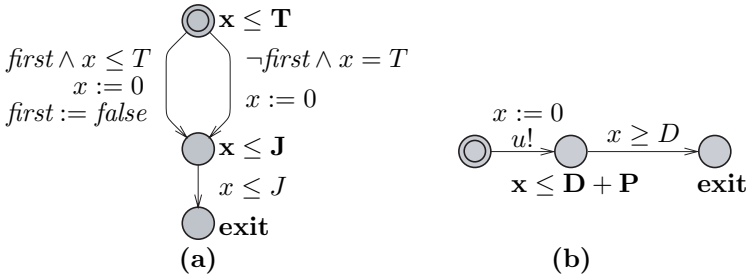

Figure 5.2: Behaviour of **(a)** Clock(T,J) and **(b)** Delay(D,P).

generating a trigger.

The behaviour automaton of a core component corresponding to *Delay(D,P)* is shown in Figure 5.2 (b). The clock x is immediately reset when the *Delay* component becomes active. When x is between D and $D + P$ time units the exit location is reached. This initiates the write phase, where the output trigger port is activated.

5.2.3 Connections

In dealing with connections, our aim has been to provide a detailed and intuitive semantics for immediate connections. Each complex connection is translated into two immediate connections with a component in between that models the behaviour of the connection. For example, a connection with a specified maximum and minimum delay *min* and *max* can be modelled using a component $Delay(min, max - min)$.

5.2.4 Connection Chains

In the full SaveCCM language, components can be connected by a chain of connections leading through several assembly ports and switches. Such chains must be collapsed into immediate, end-to-end conditional connections in the core language. Also, we should get rid of multiple output trigger ports, and combined data- and trigger ports.

Let *external_in* denote the set of external input ports, and let *external_out* denote the set of external output ports. Let $p \rightarrow p'$ denote an immediate connection from port p to port p'. For each output port p_1 of a core component C and for each $p_1 \in external\_in$, we consider all connection chains

$$p_1 \rightarrow p'_1, \quad p_2 \rightarrow p'_2, \quad \ldots, \quad p_n \rightarrow p'_n$$

such that p'_n is an input port of a core component C' or $p'_n \in external\_out$, and for each $1 \leq x < n$ we either have

a) $p'_x = p_{x+1}$ (which is the case when p'_x is an assembly port), or

b) p'_x is connected to p_{x+1} within a switch connection pattern, guarded by the condition $expr_x$.

Each such chain results in a conditional connection from p_1 to p'_n, with an expression equal to the conjunction of all switch guards in the chain (denoted $expr_x$ above).

If p_1 is a combined data and triggering port, or if p'_n is a component generated by a complex connection, then an input trigger port should be added to C' and connected to the output trigger port of C by a conditional connection with the same expression as the connection from p_1 to p'_n described above.

Appendix A

SaveCCM XML Syntax

In this appendix we describe the XML syntax of SaveCCM by describing the element definitions from the DTD (Document Type Definition).

A.1 Application

The `APPLICATION` element is the top level element of a SaveCCM XML file. The element has a unique identifier, the `id` attribute, and contains an `IODEF` element, a `TYPEDEFS` element, a `COMPONENTLIST` element, and a `CONNECTIONLIST` element.

```
⟨!ELEMENT APPLICATION (IODEF, TYPEDEFS,  COMPONENTLIST,
                                         CONNECTIONLIST)⟩
⟨!ATTLIST APPLICATION id ID #REQUIRED⟩
```

A.2 I/O Definition

The `IODEF` element contains `INPORT` and `OUTPORT` elements that define the external ports of an application.

```
⟨!ELEMENT IODEF (INPORT*, OUTPORT*)⟩
```

A.3 Type Definitions

The `TYPEDEF` element contains descriptions of components, switches and assemblies that may be used when composing the application.

```
⟨!ELEMENT TYPEDEFS (COMPONENTDESC*, SWITCHDESC*, ASSEMBLYDESC*)⟩
```

A.4 Component Description

A COMPONENTDESC element describes a component as a set of ports, attributes, and an internal behaviour. The element has a unique identifier in the id attribute, and contains INPORT elements, OUTPORT elements, ATTRIBUTE elements, a BEHAVIOUR element, and a REALISATION element. The behaviour is used to attach models to a component, for use in different tools. The realisation describes the componenet as either a C function (ENTRYFUNC), a clock/delay component, or as a composition.

```
⟨!ELEMENT COMPONENTDESC   (INPORT*, OUTPORT*, ATTRIBUTE*,
                          BEHAVIOUR, REALISATION)⟩
⟨!ATTLIST COMPONENTDESC   id ID #REQUIRED⟩
⟨!ELEMENT REALISATION     (ENTRYFUNC | CLOCK | DELAY |
                          (COMPONENTLIST, CONNECTIONLIST))⟩
⟨!ELEMENT CLOCK #PCDATA⟩
⟨!ATTLIST CLOCK period    CDATA #REQUIRED jitter CDATA #IMPLIED⟩
⟨!ELEMENT DELAY #PCDATA⟩
⟨!ATTLIST DELAY delay     CDATA #REQUIRED precision CDATA #IMPLIED⟩
```

A.5 Switch Description

A SWITCHDESC element describes a switch as a set of ports and a set of switching conditions. The element has a unique identifier in the id attribute, and contains INPORT elements, OUTPORT elements and SWITCHCONDITION elements.

The SWITCHCONDITION element describes under what conditions the ports of the switch are internally connected. The FROM and TO elements are described in Section A.15 (Connection). CONDITION is an empty element, with a setport and a value attribute signifying the condition that the setport has this value.

```
⟨!ELEMENT SWITCHDESC (INPORT*, OUTPORT*, SWITCHCONDITION*)⟩
⟨!ATTLIST SWITCHDESC id ID #REQUIRED⟩
⟨!ELEMENT SWITCHCONDITION (FROM, TO*, CONDITION* )⟩
⟨!ELEMENT CONDITION EMPTY⟩
⟨!ATTLIST CONDITION setport IDREF #REQUIRED value CDATA #REQUIRED⟩
```

A.6 Assembly Description

A SWITCHDESC element describes an assembly as a set of ports and a composition of components and connections. The element has a unique identifier in the id attribute, and contains INPORT elements, OUTPORT elements, a COMPONENTLIST element and a CONNECTIONLIST element.

```
⟨!ELEMENT ASSEMBLYDESC (INPORT*, OUTPORT*,  COMPONENTLIST,
                                            CONNECTIONLIST)⟩
⟨!ATTLIST ASSEMBLYDESC id ID #REQUIRED⟩
```

A.7 Component and Connection List

Lists of components and connections describe a composition, and are used both for an application and its assemblies.

```
⟨!ELEMENT COMPONENTLIST (COMPONENT*, SWITCH*, ASSEMBLY*)⟩
⟨!ELEMENT CONNECTIONLIST (CONNECTION*)⟩
```

A.8 Component

A `COMPONENT` element instantiates a component description (given by the `type` attribute), and has a unique identifier (the `id` attribute).

```
⟨!ELEMENT COMPONENT EMPTY⟩
⟨!ATTLIST COMPONENT type IDREF #REQUIRED id ID #REQUIRED⟩
```

A.9 Switch

A `SWITCH` element instantiates a switch description (given by the `type` attribute), and has a unique identifier (the `id` attribute).

```
⟨!ELEMENT SWITCH EMPTY⟩
⟨!ATTLIST SWITCH type IDREF #REQUIRED id ID #REQUIRED⟩
```

A.10 Assembly

An `ASSEMBLY` element instantiates an assembly description (given by the `type` attribute), and has a unique identifier (the `id` attribute).

```
⟨!ELEMENT ASSEMBLY EMPTY⟩
⟨!ATTLIST ASSEMBLY type IDREF #REQUIRED id ID #REQUIRED⟩
```

A.11 Behaviour

A behaviour is a collection of models that describe e.g. a component or a connection. A model can be an external file, or embedded as text within the element. The model has a `type` attribute, describing what type of model it is.

A predefined model type is `connection`, defining an UPPAAL model of a connection, either inline (using the XTA format) or as a model file.

```
⟨!ELEMENT BEHAVIOUR (MODEL*)⟩
⟨!ELEMENT MODEL #PCDATA⟩
⟨!ATTLIST MODEL type CDATA #REQUIRED filename CDATA #IMPLIED⟩
```

A.12 Entry Function

An ENTRYFUNC element describes a C function realizing the behaviour of a component.

```
⟨!ELEMENT ENTRYFUNC (BINDPORT*)⟩
⟨!ATTLIST ENTRYFUNC filename CDATA #REQUIRED entry CDATA #REQUIRED⟩
⟨!ELEMENT BINDPORT EMPTY⟩
⟨!ATTLIST BINDPORT port IDREF #REQUIRED argument CDATA #REQUIRED⟩
```

A.13 Input and Output Ports

The INPORT and OUTPORT elements define input and output ports. The mode attribute determines if a port is a data port, trigger port, or a combination of data and trigger. The type and value attributes are used for data ports (and combined ports) to define data type and initial value. The external attribute holds the label of an external port, defining a connection to an external entity (e.g. an I/O port or a database pointer). The setport attribute determines if the port is used in a switch condition.

```
⟨!ELEMENT INPORT   EMPTY⟩
⟨!ATTLIST INPORT   mode (data|trig|combined) #REQUIRED
                   type CDATA #REQUIRED id ID #REQUIRED
                   value CDATA #IMPLIED
                   external CDATA #IMPLIED
                   setport (true|false) "false"⟩
⟨!ELEMENT OUTPORT  EMPTY⟩
⟨!ATTLIST OUTPORT  mode (data|trig|combined) #REQUIRED
                   type CDATA #REQUIRED id ID #REQUIRED
                   value CDATA #IMPLIED
                   external CDATA #IMPLIED⟩
```

A.14 Attribute

Attrubutes are used to describe extra-functional properties of components.

```
⟨!ELEMENT ATTRIBUTE  EMPTY⟩
⟨!ATTLIST ATTRIBUTE  id CDATA #REQUIRED
                     type CDATA #REQUIRED
                     value CDATA #REQUIRED
                     credibility CDATA #IMPLIED⟩
```

A.15 Connection

The FROM and TO elements are references to connected ports. A connection has an optional BEHAVIOUR element, used to define complex connections. No

behaviour means that the connection is immediate, an empty behaviour means that the behaviour of the complex connection has not been specified (i.e. the model is incomplete).

```
⟨!ELEMENT CONNECTION (FROM, TO*, BEHAVIOUR?)⟩
⟨!ELEMENT FROM EMPTY⟩
⟨!ATTLIST FROM id IDREF #REQUIRED port IDREF #REQUIRED⟩
⟨!ELEMENT TO EMPTY⟩
⟨!ATTLIST TO id IDREF #REQUIRED port IDREF #REQUIRED⟩
```

Bibliography

[1] Åkerholm, M., J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson and M. Tivoli, *The SAVE approach to component-based development of vehicular systems*, Journal of Systems and Software (2006).

[2] Alur, R. and D. L. Dill, *A theory of timed automata*, Theoretical Computer Science **126** (1994), pp. 183–235.

[3] Amnell, T., E. Fersman, L. Mokrushin, P. Pettersson and W. Yi, TIMES*: a Tool for schedulability analysis and code generation of real-time systems*, in: *Proc. of 1*st *International Workshop on Formal Modeling and Analysis of Timed Systems*, LNCS (2003).

[4] Elmqvist, J. and S. Nadjm-Tehrani, *Safety-oriented design of component assemblies using safety interfaces*, in: *Proceedings of the International Workshop on Formal Aspects of Component Software (FACS'06)* (2006).

[5] Elmqvist, J., S. Nadjm-Tehrani and M. Minea, *Safety interfaces for component-based systems*, in: *Proceedings of the 24th International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, 2005.

[6] Fersman, E., P. Pettersson and W. Yi, *Timed automata with asynchronous processes: Schedulability and decidability*, in: J.-P. Katoen and P. Stevens, editors, *Proc. of the 8*th *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2280 in Lecture Notes in Computer Science (2002), pp. 67–82.

[7] Hansson, H., M. Åkerholm, I. Crnkovic and M. Törngren, *SaveCCM – a component model for safety-critical real-time systems*, in: *Proc. 30th Euromicro Conference*, 2004, pp. 627–635.

[8] Lundbäck, K.-L., J. Lundbäck and M. Lindberg, *Development of dependable real-time applications*, Arcticus Systems (2004).

[9] Nyström, D., A. Tesanovic, C. Norström and J. Hansson, *Database pointers: a predictable way of manipulating hot data in hard real-time systems*, in:

Proceeding of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2003), Tainan, Taiwan, 2003, p. 12.

[10] van Ommering, R., F. van der Linden, K. Kramer and J. Magee, *The Koala component model for consumer electronics software*, Computer **33** (2000), pp. 78–85.