

SaveCCM: An Analysable Component Model for Real-Time Systems

Jan Carlson^a, John Håkansson^b and Paul Pettersson^b

^a *Mälardalen University*
Department of Computer Science and Electronics
P.O. Box 883, SE-721 23, Västerås, Sweden

^b *Uppsala University*
Department of Information Technology
P.O. Box 337, SE-751 05, Uppsala, Sweden

Abstract

Component based development is a promising approach for embedded systems. Typical for embedded software is the presence of resource constraints in multiple dimensions. An essential dimension is time, since many embedded systems have real-time requirements. We define a formal semantics of a component language for embedded systems, *SaveCCM*, a language designed with vehicle applications and safety concerns in focus. The semantics is defined by a transformation into timed automata with tasks, a formalism that explicitly models timing and real-time task scheduling. A simple *SaveCCM* system with a PI controller is used as a case study. Temporal properties of the PI controller have been successfully verified using the timed automata model checker UPPAAL.

Key words: Components, Real-time or embedded components, Component specification, Formal methods, Case study.

1 Introduction

In the last few years, a number of models supporting components based development (CBD) of real-time and embedded systems have been proposed [9,14,15]. Like other component models, these models support specification of systems or applications built from (possibly adapted) existing components, as opposed to building a system from scratch. In addition, models for CBD of real-time and embedded systems must also support development of systems in which tight constraints on resource usage, real-time, and interactions with the environment must be satisfied.

To succeed with CBD, it is important that the component modelling language has a well-defined (informal or formal) semantics, allowing for reasoning

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

and analysis of a design model already in the early phases of a CBD project. For real-time systems a promising approach to provide analysis of models is to formally specify systems in a modelling language such as timed automata [2], and use an existing model-checking tool, e.g. UPPAAL [12] or Kronos [16], to validate the model by simulation, or to model-check if formally specified correctness properties of the system are satisfied or not. These tools and techniques are now powerful enough to specify and analyse some industrial systems [5,6,10,13]. However, their modelling languages do not provide much support for CBD.

In this paper we study the **SaveComp** component technology developed within the SAVE project¹, and its corresponding component modelling language called **SaveComp** Component Model or **SaveCCM** for short [1,8]. The **SaveCCM** language defines a graphical syntax and a run-time framework for **SaveCCM** systems, which has been used to illustrate different aspects of **SaveComp** and component-based architectures for real-time and embedded systems. However, a formal semantics of the **SaveCCM** language is still missing.

This shortcoming of **SaveComp** is the main focus of this paper. We contribute by defining a formal syntax and semantics for the modelling language **SaveCCM**. We first identify a small set of elements, called the *core* part of the language, which is such that all elements of the full **SaveCCM** can be defined by simple transformation steps into elements of the core part. For the elements of the core language, the syntax is defined and the semantics is given as models of timed automata (possibly with tasks [3]). For the full **SaveCCM** language, we show how its modelling elements can be defined in terms of the core language. This is often rather straight-forward, but some elements such as the so-called switches, used for dynamic addressing of data values, requires a little more attention.

The timed automata semantics of **SaveCCM** suggests that it should be possible to analyse **SaveCCM** models with a model-checking tool such as Kronos or UPPAAL. We test this approach in an experiment where the UPPAAL tool is applied to analyse a **SaveCCM** model of a simple PI-controller. We show how a **SaveCCM** model of the controller can be translated into timed automata, and that non-trivial properties about the controller model can be analysed. In particular we show that in a given environment, the controller design is schedulable, deadlock-free, and guaranteed to stabilise to $\pm 10\%$ within one second.

The paper is organised as follows: The syntax and semantics of the core language are presented in Sections 2 and 3 respectively. In Section 4 we present the full **SaveCCM** language and how it can be derived using constructs in the core language. A case-study is presented in Section 5, and we conclude the paper in Section 6.

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

Related work: Cadena [9] is a framework for modelling and analysis of component based embedded systems, supporting static analysis and model checking. Cadena uses the CORBA Component Model, and describes a discrete time semantics over the messages queued by middleware services. Unlike CORBA, **SaveCCM** restricts when and how components communicate — for example the data communicated between components is not queued.

Reo [4] is a coordination model for component composition. It defines a very flexible semantics of connectors, where component instances and connection endpoints can migrate during run-time. We assume that the behaviour of a connection can be described by a timed automaton.

Giotto [11] is a time-triggered language for programming embedded system. The language has well specified semantics, and support for dynamic mode switches. Like **SaveCCM** components, Giotto tasks interface to their environment through ports. The tasks follow a static schedule, while **SaveCCM** components can use other scheduling strategies.

An important property of composition is incrementality, meaning that the behaviour of a system is independent of the order of its integration. In [7] a layered approach is used to achieve an associative and commutative composition operator, thus ensuring incrementality. The **SaveCCM** execution model in [8] describes a component as either waiting, reading from input ports, performing internal computation, or writing to output ports. An assembly is associative and commutative, it does however not behave as a component. We therefore introduce the composite component as a composition that exhibit this component behaviour. As a consequence the composition is not associative, however its dependence on the order of integration is made explicit by the component borders separating external and internal ports.

2 **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 $trigger_out(C)$ when referring to this port.

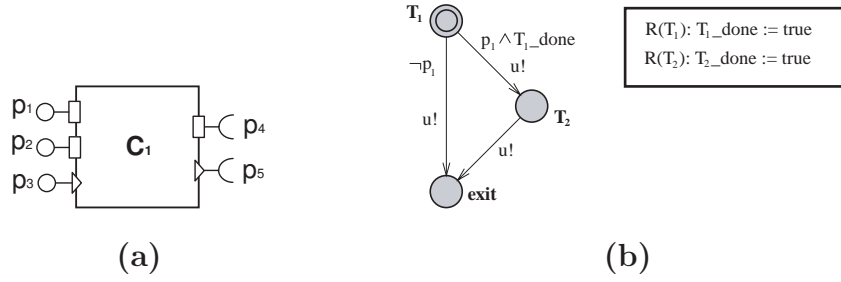


Fig. 1. (a) A basic component C_1 with three input ports and two output ports. (b) Timed automaton with tasks, describing the behaviour of component C_1 .

2.1 Basic Component

An example of the graphical syntax for basic components is shown in Fig. 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 $trigger_out(C_1) = p_5$. 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 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. 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.

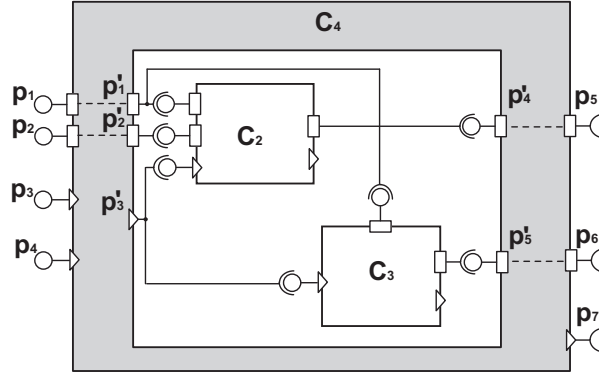


Fig. 2. A composite component composed of two internal components. The dashed lines illustrate that the internal components are not directly connected to the external ports of the composite component.

2.2 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. 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 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.

2.3 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 in Fig. 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

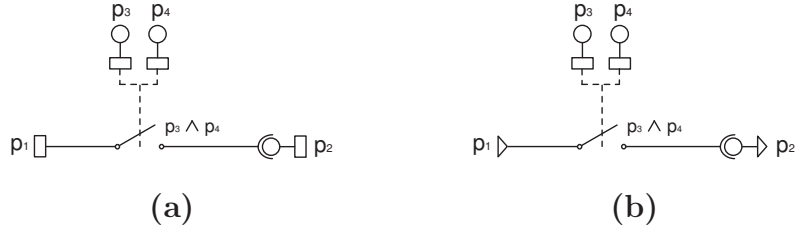


Fig. 3. A conditional connection with two setports, the connection is active when both setports are true. **(a)** connects two data ports, **(b)** connects two trigger ports.

the expression. The setports of a conditional connection are not trigger ports. The connections in Fig. 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.

3 SaveCCM Core Semantics

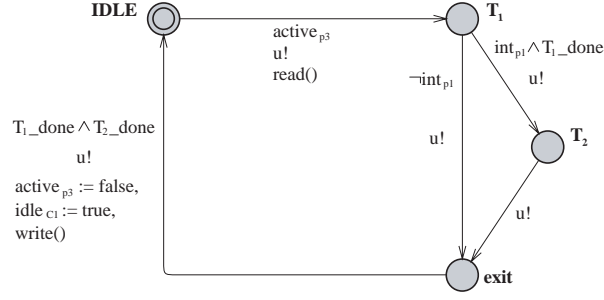
We define the semantics of SaveCCM Core by describing a translation to networks of timed automata [2] extended with tasks [3]. We extend this further with *operations*. An operation is a sequence of statements, such as variable updates or conditional **if**-statements. As mentioned above, locations can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

3.1 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


 Fig. 4. Semantics of component C_1 in Fig. 1.

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 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 shows the semantics of the component C_1 in Fig. 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 3.3. 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.

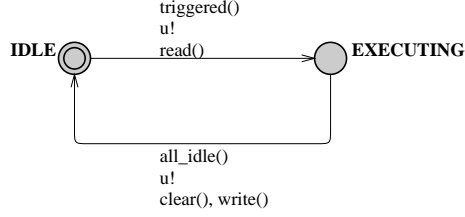


Fig. 5. Semantics of a composite component.

3.2 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. 5 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, *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_{C₂}* and *idle_{C₃}* to false by the trigger connections. When *idle_{C₂}* and *idle_{C₃}* 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_{C₁}* 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 .

3.3 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.

For the conditional connection in Fig. 3, where p_2 and p_5 are the input trigger ports of a component *C*, we define *write_x*() as in Fig. 6. If the condition $p_3 \wedge p_4$ holds, port p_2 is updated from port p_1 . For the data connection in (a),

<pre> if $ext_{p_3} \wedge ext_{p_4}$ then $ext_{p_2} := int_{p_1}$ end if </pre> <p style="text-align: center;">(a)</p>	<pre> if $ext_{p_3} \wedge ext_{p_4}$ then $active_{p_2} := true$ if $active_{p_5}$ then $idle_C := false$ end if </pre> <p style="text-align: center;">(b)</p>
---	--

Fig. 6. The $write_x$ operation for the conditional connections in Fig. 3 (a) and (b).

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.

4 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.

The PI controller depicted in Fig. 7 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.

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 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.

In addition to basic and composite components, there are two more component types in the full SaveCCM language. *Switches* are lightweight components

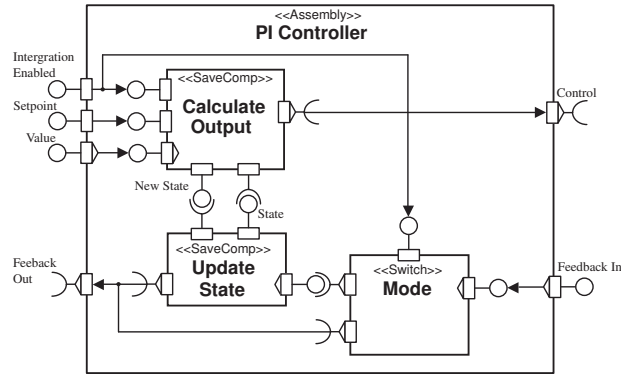


Fig. 7. An example assembly for a PI controller.

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.

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**.

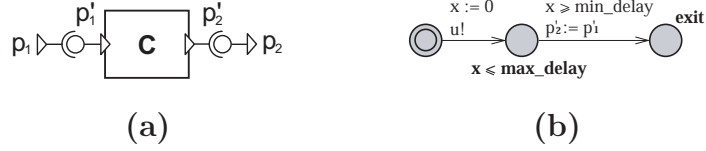


Fig. 8. **(a)** Translation of a delayed connection from p_1 to p_2 . **(b)** The behaviour automata of C .

4.1 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.

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, the translation of a connection with a specified maximum and minimum delay is depicted in Fig. 8.

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 top_in denote the set of input ports at the top system level, i.e., those that should be visible to the surrounding environment, and let top_out denote the set of top level 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 $p_1 \in top_in$, we consider all connection chains

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

such that p'_n is an input port of a core component C' or $p'_n \in top_out$, 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).

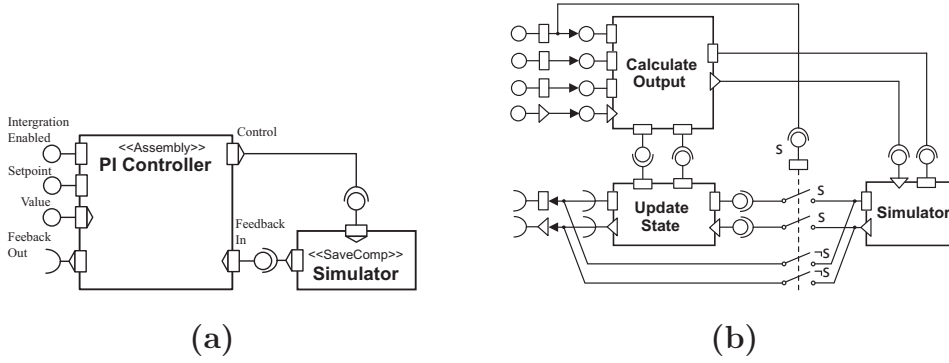


Fig. 9. A SaveCCM system (a) and the corresponding SaveCCM Core system (b).

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.

5 Case Study: A PI Controller

To illustrate the transformation described in Section 4, we show how the SaveCCM system in Fig. 9 (a) is transformed into the core system in (b). The system consists of the PI Controller shown in Fig. 7, and a component Simulator that simulates a 10cm high tank, with a constant flow of 10cm/s from the tank, and a variable flow into the tank. The input flow is actuated by the PI controller, and is limited to 15cm/s. In Fig. 9 (b) we use s as an alias for the setport Integration Enabled.

The Calculate Output, Update State and Simulator components are translated into basic core components. Ports with combined data and triggering are split into separate data and trigger ports.

We now consider the connection from Simulator to the port Feedback In of the PI controller. This connection is forwarded within the assembly to the switch Mode. From Mode, depending on its setport there is either a connection to Update State or to Feedback Out. We thus have two chains of connections, one from Simulator to Update State, and one from Simulator to the port Feedback Out. Since these chains connect ports that combine data and triggering, each of them is translated into two conditional connections, one for triggering and one for data. The conditions associated with these conditional connections are s for the two connections to Update State, and $\neg s$ for the other two. Other connection chains do not pass through a switch, so the condition is true for their conditional connections.

If we were to introduce a delay in the connection from PI Controller to Simulator, a basic component would be inserted into Fig. 9 (b) between the components Calculate Output and Simulator. If we introduce delay in the

reverse direction, from **Simulator** to **PI Controller** as well, this would result in another basic component positioned between **Simulator** and the conditions.

5.1 Analysing the PI Controller

Before we analyse the PI Controller, we need to setup an environment providing data and triggering. We use the feedback ports so that the simulator can provide sensor data to the port **Value**. A timed automaton is used to periodically copy sensor data along with triggering every 10ms. The ports **Setpoint** and **Integration Enabled** are set to 5cm and **true**, respectively.

We analyse the system in UPPAAL which does not support floating point data, so we use a fixed point representation with two decimal places. Time is measured in ms. Verifying that after one second the controlled value becomes stable within 10% of the setpoint took 2.3s and 20Mb on a 1.66GHz Intel Celeron. This stability property is expressed as:

$$A\Box(now \geq 1.0s \Rightarrow 0.9 \leq Value/Setpoint \leq 1.1)$$

The clock *now* measures time elapsed since the initial state, while the variables *Value* and *Setpoint* represent the current sensor value and desired value, respectively. The PI controller was setup with a proportional constant $K = 1.66$, and an integration time $T_i = 0.15s$.

Other properties such as deadlock freedom and schedulability were verified using less resources. The stability property requires more resources since it introduces a clock constraint with the constant 1.0s, represented as integer 100,000. The algorithm for model checking timed automata depends on the largest constant used in a clock constraint.

6 Conclusions

We have defined a formal semantics of **SaveCCM** by providing a translation to an intermediate core language, and by mapping constructions of the core language to timed automata with tasks. The formal semantics is such that the switch construction has the same semantics when replaced with immediate connections for a static configuration. This was a goal since the switch was intended to be used for both static (configured before run-time) and dynamic (run-time) configuration. We have also shown how a simple PI controller can be translated to the core language, and that non-trivial properties of the resulting model can be analysed using UPPAAL².

TIMES³ is a tool for modelling, simulation, and analysis of timed automata extended with tasks. The TIMES tool has extensive support for schedulability analysis, however the current version has no support for operations. A new version is planned with support for the same C-like syntax as used in the de-

² UPPAAL is available from www.uppaal.com

³ The TIMES tool is available from www.timestool.com

velopment versions of UPPAAL (3.5.x). Using this tool it might be possible to perform schedulability analysis of more detailed *SaveCCM* models. Another direction for future work is to take advantage of the read–execute–write semantics of components during analysis, and use partial order reduction techniques to reduce the size of the state-space analysed by a model-checker.

References

- [1] Åkerholm, M., A. Möller, H. Hansson and M. Nolin, *Towards a dependable component technology for embedded system applications*, in: *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2005)* (2005).
- [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 1st International Workshop on Formal Modeling and Analysis of Timed Systems*, LNCS (2003).
- [4] Arbab, F., *Reo: a channel-based coordination model for component composition*, *Mathematical Structures in Comp. Sci.* **14** (2004), pp. 329–366.
- [5] Bengtsson, J., W. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson and W. Yi, *Verification of an Audio Protocol with Bus Collision Using UPPAAL*, in: *Proceedings of CAV’96*, LNCS (1996), pp. 244–256.
- [6] David, A. and W. Yi, *Modelling and analysis of a commercial field bus protocol*, in: *Proceedings of the 12th Euromicro Conference on Real Time Systems* (2000), pp. 165–172.
- [7] Gössler, G. and J. Sifakis, *Composition for component-based modeling*, in: F. de Boer, M. Bonsangue, S. Graf and W.-P. de Roever, editors, *Proceedings of the 1st Symposium on Formal Methods for Components and Objects (FMCO 2002)*, LNCS **2852** (2003), pp. 70–98.
- [8] Hansson, H., M. Åkerholm, I. Crnkovic and M. Törngren, *SaveCCM - A component model for safety-critical real-time systems*, in: *Proc. of Euromicro Workshop on Component Models for Dependable Systems* (2004).
- [9] Hatcliff, J., W. Deng, M. Dwyer, G. Jung and V. Prasad, *Cadena: An integrated development, analysis, and verification environment for component-based systems*, in: *Proceedings of the 25th International Conference on Software Engineering*, 2003.
- [10] Havelund, K., A. Skou, K. G. Larsen and K. Lund, *Formal modelling and analysis of an audio/video protocol: An industrial case study using UPPAAL*, in: *Proceedings of the 18th IEEE Real-Time Systems Symposium*, 1997, pp. 2–13.

- [11] Henzinger, T. A., B. Horowitz and C. M. Kirsch, *Giotto: A time-triggered language for embedded programming*, LNCS **2211** (2001), pp. 166–184.
- [12] Larsen, K. G., P. Pettersson and W. Yi, *UPPAAL in a Nutshell*, Int. Journal on Software Tools for Technology Transfer **1** (1997), pp. 134–152.
- [13] Lindahl, M., P. Pettersson and W. Yi, *Formal Design and Analysis of a Gearbox Controller*, Int. Journal on Software Tools for Technology Transfer **3** (2001), pp. 353–368.
- [14] Stankovic, J. A., *VEST - A toolset for constructing and analyzing component based embedded systems*, in: *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software* (2001), pp. 390–402.
- [15] van Ommering, R., F. van der Linden, J. Kramer and J. Magee, *The Koala component model for consumer electronics software*, IEEE Computer (2000), pp. 78–85.
- [16] Yovine, S., *KRONOS: A verification tool for real-time systems.*, Int. Journal on Software Tools for Technology Transfer **1** (1997), pp. 123–133.