A component-based development framework for supporting functional and non-functional analysis in control system design*

Johan Fredriksson
Mälardalen University
Mälardalen Real-Time Research Centre
Västerås, Sweden
johan.fredriksson@mdh.se

Massimo Tivoli
University of L'Aquila
Computer Science Department
L'Aquila, Italy
tivoli@di.univaq.it

Ivica Crnkovic
Mälardalen University
Mälardalen Real-Time Research Centre
Västerås, Sweden
ivica.crnkovic@mdh.se

Abstract

The use of component-based development (CBD) is growing in the software engineering community and it has been successfully applied in many engineering domains such as office applications and in web-based distributed applications. Recently, the need of CBD is growing also in other domains related to dependable and embedded systems, namely, in the control engineering domain. Control systems constitute the core functionality of modern embedded systems such as vehicles and consumer electronics. However, the widely used commercial component technologies are unable to provide solutions to the requirements of embedded systems as they require too much resource and they do not provide methods and tools for developing predictable and analyzable embedded systems. There is a need for new component-based technologies appropriate to development of embedded systems.

In this paper we present a component-based development framework called SAVEComp. SAVEComp is developed for safety-critical real-time systems. One of the main characteristics of SAVEComp is syntactic and semantic simplicity which enables a high analyzability of properties important for embedded systems. By means of an industrial case-study, we show how SAVEComp is able to provide an efficient support for designing and implementing embedded control systems by mainly focusing on simplicity and analyzability of functional requirements and of real-time and dependability quality attributes. In particular we discuss the typical solutions of control systems in which feedback loops are used and which significantly complicate the design process. We provide a solution for increasing design abstraction level and still being able to reason about system properties using SAVEComp approach. Finally, we discuss an extension of SAVEComp with dynamic run-time property checking by utilizing run-time spare capacity that is normally induced by real-time analysis.

*This work is an extended and revisited version of [18].

1 Introduction

The use of component-based development (CBD) is growing in the software engineering community and it has been successfully applied in many engineering domains such as desktop environments, office applications, e-business and in web-based distributed applications. To improve control systems analysability, reusability, flexibility and to decrease the *time-to-market*, the need of CBD is growing also in other domains related to dependable and embedded systems (i.e., control engineering domain). Control systems constitute the core functionality of modern embedded systems such as vehicles and consumer electronics. One of the main issues in control engineering domain is how to design control systems in such a way that functional requirements (safety and liveness properties) as well as real-time attributes (end-to-end timing, freshness of data, simultaneity, jitter tolerances, WCET) can be analyzed already in an early phase of the system life-cycle, namely during design-time.

Due to the increasing complexity of control systems, they are often constructed performing a modular approach by means of libraries of building blocks with high functionality and a high degree of flexibility. This has lead to a need of a component-based approach for building control systems out of a set of already implemented *"control modules"* [15]. The control module concept has been implemented in *ABB's new control system, Control IT* as a more reliable and *easy-to-use* generalization of a traditional IEC61131-3 function block[1] [1]. A control module might be considered a control system component and hence it is the mean to build control systems by adopting a component-based approach supported by a suitable component technology. Unfortunately, commercial component technologies are too complex and unpredictable and hence, predictability of the functional and non-functional behaviour of the system would be weakly supported and in most cases not supported at all. Moreover, although component models that support predictability of the system behaviour exist, they are often not able to support the requirements of embedded systems. For example, software components for embedded systems should provide an interface specification that points out specific resource requirements or other properties of interest for the target application, e.g., timing, memory usage and dependability-related attributes such as reliability and safety. Specific architectural constraints should be imposed on the system design in such a way that predictability of properties that are relevant for the domain can be supported. Even a component framework for embedded systems should use predictable mechanisms and be light weight. Thus, a component-based development framework which supports the requirements of embedded systems

[1] In the reminder of the paper, we will use the term "function block" to identify a "IEC61131-3 function block" and all its further extensions (e.g., IEC61499 function blocks [9]).

is highly needed in order to be able to predict functional and non-functional behaviour of control systems during design-time.

In this paper, we present a component-based development framework, called *SAVEComp*[2] that supports predictability of control system behaviour during design-time. The main purpose of SAVEComp is to provide efficient support for designing and implementing embedded control applications by mainly focusing on simplicity and analysability of functional requirements and real-time and dependability properties. Our reference component model is *SaveComp Component Model* (SaveCCM) [8] which is designed for safety-critical real-time systems. SaveCCM has been thought to support predictability of the real-time behaviour of embedded systems. We show how to extend the current version of SaveCCM in order to incorporate the control module concept in SAVEComp in such a way that we are able to predict the system behaviour. A control module in SAVEComp is inherently able to correctly deal with outer and inner control loops that are typical of control systems, where control flow feedbacks must be handled to deliver the response for the time-critical computation as fast as possible. By exploiting the existent architectural elements of SaveCCM, we can define a control module as a new composite architectural element that - when composed with other control modules to build control loops - satisfies requirements that are needed for the correct functioning of the control system and to predict its behaviour. For example, the SaveCCM control modules within a control loop satisfy that the backwards flow is always executed only after the forwards flow has been completely performed. Moreover, the design of a SaveCCM control module can be enriched with information about the module quality attributes by providing the ground support for the system analysis. By means of both the extended capabilities of SaveCCM and the analysis tools provided by SAVEComp, we show how the developer is able to build control systems by composing already implemented components in such a way that both functional requirements and real-time attributes can be analyzed in control systems design. We also discuss an extension of SAVEComp with dynamic run-time property checking by utilizing run-time spare capacity that is normally induced by real-time analysis. We validate the applicability of SAVEComp and its appropriateness for the domain of embedded systems by means of an industrial case study.

The remainder of the paper is organized as follows. Section 2 discusses related work and a brief comparison between SaveCCM and other component-technologies. Section 3 discusses background notions of our work by referring to control modules as a solution for an "easy-to-make" component-based design of control systems. In Section 4 the main features of SaveCCM are summarized. In Section 5 we first outline the overall structure of SAVEComp and then - by means of an explanatory example - we discuss its relevant aspects in more detail. Section 6 validates the applicability and appropriateness of SAVEComp for the embedded systems domain by means of an industrial case study that is concerned with an adaptive cruise controller. Section 7 concludes and discusses 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. We will also discuss smilarities with ADLs like, e.g., Darwin.

[2]SAVEComp is developed in the project *SAfety critical components for VEhicular systems - http://www.mrtc.mdh.se/SAVE*.

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 nonfunctional 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 realtime 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 independences 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 generalpurpose 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.

Darwin is a general declarative ADL for distributed software architectures. Differently form SaveCCM, Darwin provides one with primitives of a *"pure"* structural language since the underlying concepts of components and binding are independent from the interaction mechanisms between components. Thus, Darwin is more general and can be applied to more conventional program structures. On the other hand, SaveCCM has been designed to represent a specific real-time component model. Although the domain-specific approach means loss of generality, it has the advantages of simpler and precise expression of the design, better design comprehension, and support of automatic analysis and verification. Moreover, Darwin does not support the specification of non-functional properties and, hence, real-time attributes analysis would be unfeasible in using the Darwin component model without a applying a suitable extension of it aimed at supporting real-time properties.

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 componentbased 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 Background: Control Modules

In Section 1, we said that many modern control systems are designed by using a modular approach in which its constituent function blocks are combined together.

Function blocks are very complex and have many configuration parameters because the rapid development of control algorithms has lead to a tremendous increase of the function block's functionalities. There are two main disadvantages due to the increased complexity of the function blocks. The first one is that there are a lot of parameters to be set and interface points to be connected and, hence, the developer should have a deep knowledge of the different function blocks. The second one is the obvious risk to make mistakes when the developer has to deal with a large amount of parameters and interface points. In [15], a component-based solution to overcame these disadvantages has been proposed. The main idea is to reduce the complexity of control systems by defining a standard interface for the signals between the building blocks. This implies that the blocks have to be constructed according to component-oriented principles (as we will see later each one of them can be constructed as an aggregate component in our reference component model). A **ControlConnection** data structure which allows one to connect these building blocks has been defined in [15]. This data structure contains all the signals that are sent between the function blocks of the control system. Since real-scale control systems, often, have to deal with control loops, some of the signals are sent forwards and some are sent backwards; thus, ControlConnection collects all the signal in two substructures called **Forward** and **Backward** respectively.

In Figure 1.**A** we show an example of a control system made of a cascade control loop [14] where its building blocks are traditional function blocks. In Figure 1.**B** we show the same cascade control loop where its building blocks are connected by means of a graphical connection of ControlConnection type. Note that a control system is configured in a much simpler way if the blocks are connected with a ControlConnection structure.As showed in the figure, we will hereafter refer to the simpler configuration as the *top-level* design of the control system and to the other one as its *internal design*. In order to deal with connections of ControlConnection type, all the building blocks of the loop have to be able to transmit information forwards as well as backwards, with low delays. For this reason, in [15], the concept of control module has been introduced as a generalization of a traditional function block. The control module contains two parts of code for transmitting information forwards and backwards respectively. Although the control module concept considerably reduces the complexity of control loops by providing a component-based approach, current component technologies do not allow one to realize a control module in order to provide the developer with facilities for supporting predictability of the control system behaviour. This leads to a real need of a component-based approach for designing and composing control modules in such a way that such a support can be provided. Our aim is to provide a mean that will make it possible to use a component-based approach and predict the system behaviour.

4 The SaveCCM component model

In this section we briefly describe the main characteristics of our reference component model called SaveCCM. Refer to [8] for a detailed description of it.

The SAVEComp Component Model (SaveCCM) [8, 2] is a restrictive component model for control software development. It consists of the following main architectural elements: components, switches, assemblies and *run-time framework*, which provides a set of services, such as communication between components, component execution and control of sensors and actuators.

The interface of an architectural element is defined by a set of ports, i.e., points of interaction between the element and its environment. SaveCCM distinguishes 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 distinguishes between these two aspects, and allows three types of ports: (i) *data-only* ports, (ii) *triggering-only* ports, and (iii) *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 by composing architectural elements. This composition is obtained by connecting input ports to output ports.

Since predictability and analysisability are of primary concern for the considered application domain, the SaveCCM execution model is rather restrictive. The basis is a control-flow (i.e., pipes-and-filter) paradigm in which executions are triggered by clocks or external events, and where components have finite, possibly variable, execution time. At the beginning a component is in an *idle* state where it waits for the activation of all its triggers. Once all component triggers have been activated, the component reads its input ports (*reading* state), performs its computations (*executing* state) based on the inputs read and its internal state, writes the result of the execution on its output ports (*writing* state) and finally goes back to the *idle* state. A list of quality attributes and (possibly) their value and credibility (i.e., a measure of confidence of the expressed value) is included in the specification of components and assemblies. In this paper we will only consider real-time attributes. We will show how such attributes can be specified and used in analysis. Component behaviour is defined by means of variables that express internal states, and actions that describe the component execution. Actions are abstract specifications of the externally visible behaviour of the component. Components are specified by their interfaces, behaviour and quality attributes.

A subset of the UML2 component diagrams[3] is adopted as graphical specification language[4]. The symbols showed in Figure 2 are used.

5 The SAVEComp development framework

In this section we outline the overall structure of the SAVEComp development framework[5] (see Figure 3). SAVEComp implements the approach we present in the following subsections as one part of its overall structure. SAVEComp has been thought to be an extensible component-based development framework for design-time analysis (both functional and non-functional) and development of

[3] UML2.0 specification - http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML.

[4] In [8], the complete textual syntax (i.e., BNF specification) of the specification language is reported.

[5] The framework is under construction

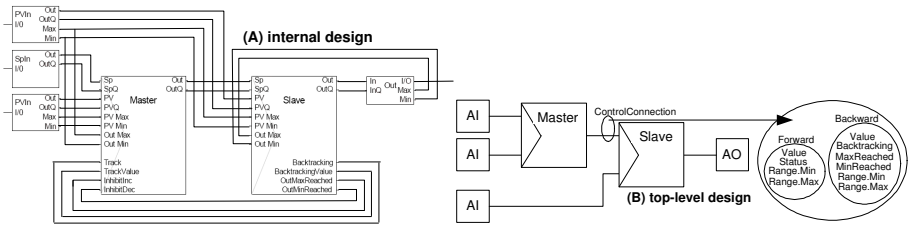

Figure 1. Two different designs of the same control system

Symbol	Interpretation	Symbol	Interpretation
	**Input ports** - The upper symbol 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 one is an input port with data and triggering.	<<SaveComp>> **<name>**	**Component** - A component with the stereotype changed to <<SaveComp>> corresponds to a SaveCCM component.
		<<Switch>> **<name>**	**Switch** - Components with the stereotype <<Switch>>, correspond to switches in SaveCCM.
	**Output ports** - Analogously to the input ports, the upper symbol is an output port with a trigger, and no data. The middle symbol is an output port with data and no triggering, and the lower one is an output port with data and triggering.	<<Assembly>> **<name>**	**Assembly** - Components with the stereotype <<Assembly>>, correspond to assemblies in SaveCCM.
			Delegation - A delegation is a direct connection from an input to input or output to output port, used within assemblies.

Figure 2. The SaveCCM graphical specification language

safety-critical embedded real-time systems. A part of it is the AutoComp technology [17] which is intended only for predicting the real-time behaviour of the system.

As showed in Figure 3, SAVEComp can be described by distinguishing three main phases of its utilization. During design-time, developers may exploit the new capabilities of SaveCCM - we present in the following subsections - to specify the top-level design of the control system by adopting a component-based software engineering process Moreover, the extended version of SaveCCM allows the developer to enrich the system design with: (i) functional properties of the system expressed in terms of sequences of actions performed on component ports and/or possible values of data ports of interest for the analysis (e.g., the set of possible values of a data port expressing different operational modes of the control system); and (ii) high level temporal constraints in form of end-to-end deadlines and jitter supplied with their credibility values. During compile-time, SAVEComp automatically produces the SaveCCM internal design corresponding to the top-level and derives different views of the designed system intended to support both different kinds of specific functional/non-functional analysis and the mapping process to a real-time operating system (RTOS). In the figure, we show two possible classes of system views/models: (i) behavioural models (e.g., Process Algebras, LTSs, state machines, MSCs, UML2 interaction diagrams); and (ii) real-time models (e.g., Worst-case execution time analysis and Response-time analysis). The first class is intended to perform functional analysis (i.e., checking safety and liveness properties[6]), the second one to perform non-functional analysis in the specific case of guaranteeing real-time attributes. The plug-in based nature of SAVEComp allows us either to add new classes of system models - whenever it is needed to perform other specific kinds of analysis - or to extend an existent class to contain other model notations that are needed to support/integrate other processes for the same kind of analysis. For example, as sketched in the figure, we might need to add a probabilistic models view (e.g., Markov Chains, Stochastic Process Algebras) to perform reliability analysis by taking into account, e.g., the credibility value of each real-time attribute. Each specific kind of analysis/transformation is supported by a plug-in based tool within SAVEComp. Each "plug-in" might be either an existent tool suitably integrated with SAVEComp or built from scratch. By looking at the result of each particular analysis, the developer can either refine the top-level design since a functional or non-functional requirement has not been met or - if the design matches every requirement - execute a synthesis step. In each utilization phase, the developer has the possibility to interact with a particular plug-in based tool to set specific configuration parameters of it or to apply refinements (that are dictated by the analysis results) directly on the generated data/models rather than being forced to go back to the original design. We choose *Eclipse* platform[7] as implementation environment since it provides us with all the integration features we need to build SAVEComp. Eclipse facilitates the integration of different tools, that usually manipulate different content types. SAVEComp is built on a XML-based core which isthe substrate providing an intermediate XML-based representation of system models that may work as a common ground to apply functional and non-functional analysis. To make SAVEComp as extensible as possible the XML core is kept general enough to allow its further extensions needed to manage new system model notations and new analysis processes and tools. In the reminder, we will only focus on the parts of SAVEComp that implement the approach presented in this paper. We consider the following SAVEComp plug-ins:

SaveCCM Visual Editor. A visual editor supporting the SaveCCM graphical specification language for designing the system architecture and for specifying functional properties and real-time attributes that must be analyzed. It is also responsible for generating the XML code that represents the SaveCCM textual specification of the designed system. Moreover, it provides the developer with compiling functionality to, e.g., perform a type-check on the component connections (e.g., a port of a component can be connected to a port of a different component only if these ports have the same type).

SaveCCM top-level (to internal) Design Converter. We recall that, control loops are often used to deliver the response for

[6] As usual, for safety and liveness we mean *nothing bad happens* and *something good eventually happens*, respectively.

[7] The Eclipse project. Eclipse platform technical overview. Technical report, 2001 - http://www.eclipse.org.

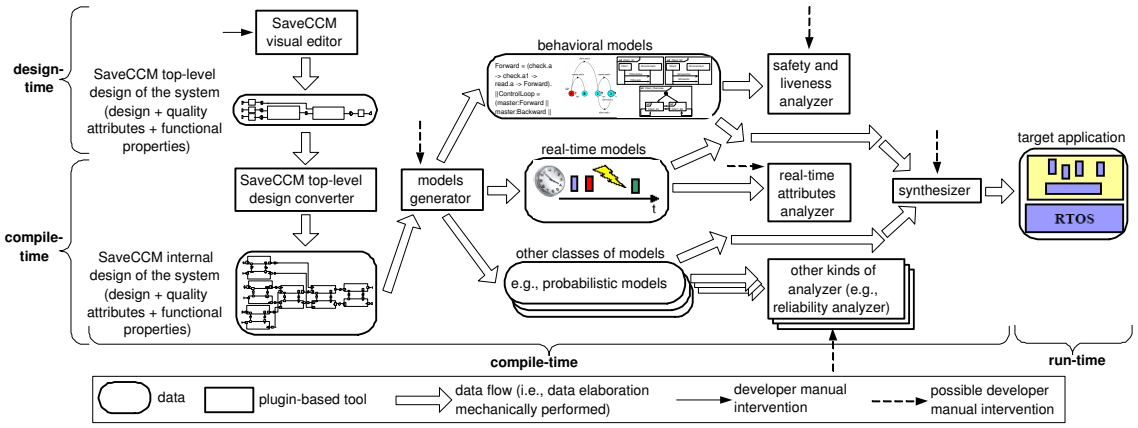

Figure 3. The SAVEComp development framework

the time-critical computation as fast as possible. Due to the increasing complexity of control loops, it is hard to correctly design them, and they might limit the predictability of the control system. As we will see in the following sections, to address these problems, control loops are designed - in SaveCCM - by means of assemblies and connections of *ControlComponent* and *ControlConnection* type, respectively. A ControlComponent implements the concept of control module by providing the developer with a higher level of abstraction in designing control loops. Thus, a control loop in a SaveCCM control system has a top-level design. This plug-in is responsible for automatically deriving from the top-level design its corresponding internal design consistent of SaveCCM components, switches and their connections. Since top-level components, as SaveCCM assemblies, do not reflect the execution model of a basic component, this translation is required in order to perform functional and non-functional analysis of the system. The translation algorithm exploits the implicit internal structure and semantics of a ControlComponent and a ControlConnection.

Functional behaviours Models Generator. A part of the *models generator* plug-in based tool. It is responsible for generating models of the functional behaviour of the designed system. The kind of generated model (e.g., Process Algebras, LTSs, state machines, MSCs, UML2 interaction diagrams) depends on the XML template used - during design-time - to specify the system's functional properties that must be checked. Once the kind of model that must be generated is established, the model is generated by taking into account the system's internal design, the execution model of the SaveCCM components forming the system, the set of possible actions performable on a SaveCCM port and its possible values. Furthermore, a consistent model (with respect to both the notation used to model the system's functional behaviour and the analyzer that will be used) of the functional properties is generated.

Safety and Liveness Analyzer. It is a plug-in based tool integrating an analyzer for each kind of model of the system's functional behaviour that can be generated. By exploiting the generated models of the system and of the properties that must be checked, the developer can interact with the suitable analyzer in order to mechanically verify the specified safety and liveness properties. For example, the developer can verify that deadlocks do not occur or that the system always progresses (i.e., can every action eventually be performed?) or other specific functional properties of the system (e.g., a specific component must be disabled if the system is running in a specific operational mode).

Component to Task Converter. A part of the *models generator* plug-in based tool. In cooperation with the *Task Attribute Assignment*, it is responsible for generating a real-time model. The algorithm strives to reduce the number of operating system tasks by allocating components to the same task according to a set of rules, e.g., when components execute with the same period-time or are triggered by the same event. The task allocation approach utilizes stochastic search techniques to find allocations that are optimized considering user-specified properties, e.g., low context-switch overhead or low memory usage.

Task Attribute Assignment. It is part of the *models generator* plug-in based tool. In cooperation with the *Component to Task Converter*, it assigning attributes considering platform and analysis goal.

Real-Time Analyzer. The analysis step is dependent on the underlying platform, e.g., schedulability analysis is limited to the algorithms available in the OS used. In the current prototype implementation, response-time analysis according to FPS theory is performed. If the response-time analysis fails, the affected parts of the system are highlighted. For instance it may be possible to find another allocation from components to tasks that satisfies the given real-time constraints. Otherwise the design may have to be revised.

Code Synthesizer. The code generation module of the compile-time activities generates all source code that is dependent on the underlying operating system. Each operating system needs to have a transformation API where platform independent system calls can be translated to OS specific. Such layers can easily be derived from reference manual of a specific RTOS. Within this step the binary representation of the system is created, often the operating system and the run-time framework are also included with the application code in a single bundle.

5.1 Extending SaveCCM to design and use control modules

The control module concept can be implemented in SaveCCM by means of a new type of assembly which composes two components. We denote this new assembly type as "ControlComponent" type. One component within a ControlComponent is denoted as "Forward", the other one is denoted as "Backward". Forward and Backward are for transmitting information forwards and backwards (within a loop in a control system), respectively. In other words, Forward is responsible - given input values and taking into account

the state of its ControlComponent - for calculating the output value of the ControlComponent. Analogously, Backward is responsible for updating the state of its ControlComponent depending on the feedback signals. Forward exports an interface made of input and output data-and-triggering ports and, possibly, other ports explicitly specified by the developer for specific purposes depending on the system functionality. The same is for Backward. ControlComponent, in turn, exports the same interface of Forward and Backward. As it is usual in SaveCCM, the ports of ControlComponent are connected to the corresponding ports of Forward and Backward through delegation. In Figure 4, we show both the SaveCCM top-level design of a ControlComponent (i.e., left-hand side) and its internal design (i.e., right-hand side). In the figure we show also labels that are used to refer the I/O ports. They model port names and they are specified only internally and do not appear at design level.

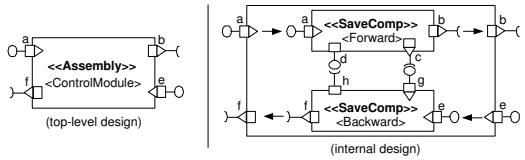

Figure 4. Top-level and internal design of "ControlModule"

It is worth mentioning that Forward and Backward, as usual SaveCCM components, respect the component execution model mentioned in Section 4. Since a ControlComponent is an assembly in SaveCCM, it is not subject to the rules of the execution model of a SaveCCM component. In other words, a SaveCCM assembly is only intended for design purposes[8] (i.e., for modelling a collection of components and hiding the internal structure rather than for component composition) and when we want to reason about its execution model we have to refer to its internal structure. The type of a data transmitted through a port of the ControlComponent is a structured data type as defined by the ControlConnection structure. The triggering data are used for activating a Forward or Backward component depending on the control flow of the system. The information required to update the state of all the ControlComponents in a loop is not available until all the Forward components have executed their code. This is required for a correct functioning of the control system. Note that a ControlComponent can handle outer control loops as well as inner loops. An inner control loop can be performed by means of the inner connections among Forward and Backward (i.e., "c", "g" and "h", "d" port connections). These inner connections are internally generated - after the generation of Forward and Backward - by the *"SaveCCM top-level design converter"* (see Figure 3). So far, we just have presented the structure of a control module as it can be built in SaveCCM. To be able to specify a top-level design, we have to be able to connect, e.g., two ControlComponent by means of a connection of ControlConnection type. Thus we have to show how to build a ControlConnection in SaveCCM. The next subsection has been intended for this purpose.

5.2 Extending SaveCCM to compose control modules

For our purposes, we extend the set of SaveCCM port types by adding a port of "Control" type. A Control port is allowed only on the functional interface of a ControlComponent. In the left-hand side of Figure 5 we show both the top-level design of a Control port and its internal design.

[8]Assemblies are really useful, e.g., for identifying patterns of aggregates of component instances that serve for providing some high-level functionality.

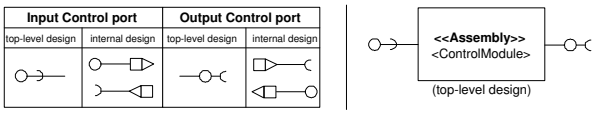

Figure 5. Top-level and internal design of a Control port and final top-level design of "ControlModule"

Note that - internally - a Control port is a bidirectional one. We distinguish between input and output Control ports. When an input Control port is attached to a ControlComponent - internally - the *"SaveCCM top-level design converter"* produces: (1) an input and an output data-and-triggering port on the ControlComponent (i.e., "a" and "f" in Figure 4); (2) an input data-and-triggering port on Forward (i.e., "a"); and (3) an output data-and-triggering port on Backward (i.e., "f"). Finally, the input data-and-triggering port of the ControlComponent is associated - through delegation - with the corresponding one of Forward. Analogously, the output data-and-triggering port of the ControlComponent is associated to the corresponding one of Backward. When an output Control port is attached to a ControlComponent, the design converter behaves analogously. By means of Control ports, the top-level design of "ControlModule" (showed in Figure 4) looks as it is showed in the right-hand side of Figure 5.

5.3 Analyzing functional requirements

In this section we formalize the execution model of a ControlComponent. This formalization is intended to support functional analysis of control systems during design-time. We are interested in proving safety and liveness properties. To formalize the execution model of a ControlComponent we look at (i) its internal design; (ii) the execution model of a SaveCCM component; (iii) the set of possible actions performable on a SaveCCM port and, in some cases[9], (iv) its possible values. By referring to Section 4, the execution model of a component may be expressed as a combination of actions that can be executed on its ports. The only action that can be performed on an input (output) data port is a reading (writing) action. We denote it as "read" ("write"). "read" and "write" are non-blocking actions (i.e., there will always be a value on a data port and it will always be possible to overwrite that value). On an input (output) triggering port we can perform a checking (activating) action that we denote as "check" ("activate"). "check" is a blocking action, that is it makes a component waiting for the activation of an input triggering port. "activate" simply activates the trigger associated to an output triggering port. On an input (output) data-and-triggering port a component executes "check" followed by "read" ("write" followed by "activate"). These rules can be combined in the obvious way in order to specify the execution behaviour of a component, with an arbitrary number of ports of different type, by means of a process algebra. Note only that if a component C has $p_1, ..., p_n$ input data-and-triggering ports then - during the initial part of its execution - C will execute a sequence of n "check" (each of them for each p_i) followed by a sequence of n "read". We choose FSP [10] (*Finite State Processes*) as process algebra to model the execution behaviour of components and assemblies at design level. FSP fits our purposes because it is notoriously easier to use than other more expressive process algebras and it is supported by LTSA [10] (*Labelled Transition System Analyser*). LTSA is a plug-in based verification tool for concurrent systems. It mechanically checks that the specification of a concurrent system satisfies

[9]This is required only for specific data ports of interest, e.g., boolean data ports used to set different operational modes of the control system.

required properties of its behaviour. In addition, LTSA supports simulation to facilitate the interactive exploration of the system behaviour. Thus the FSP specification of a SaveCCM system represents the mean to integrate SAVEComp with LTSA in order to support functional analysis. In Figure 6.**A** we show the top-level design of the control system - showed in Figure 1 - as specified by the developer using the *"SaveCCM visual editor"*[10], its internal design (Figure 6.**B**) as mechanically derived by the *"SaveCCM top-level design converter"*, its FSP specification (Figure 6.**C**) and an its liveness property (Figure 6.**L3**) that we want to verify.

The FSP specification has been mechanically derived by the *"functional behaviours model generator"* taking into account the loop's internal design, the execution model of a SaveCCM component and by combining the above mentioned rules (defining the set of actions that can be performed on a port) in the obvious way. **L3** has been included in the system top-level design (in a XML format) and it has been mechanically translated in the LTSA property notation by the *"functional behaviours model generator"*. Integrating SAVEComp with LTSA (i.e., a possible *"safety and liveness analyzer"*) allow us to easily verify functional properties of the system's FSP specification. For example, we can mechanically verify that deadlocks do not occur in the execution of the control system (i.e., safety). Moreover, we can also verify that the execution of the control system holds the liveness property showed in the figure. In Figure 6.**L3** we show the graphical notation used by LTSA to express a liveness property. It is given in form of its Büchi Automaton [4]. Informally, the Büchi Automaton is an operational description of the property and specifies the set of system behaviours that hold it. 0 denotes the initial state. 3 denotes the accepting state. *E* is an error state (i.e., a non-accepting sink node). Each arc label denotes a possible action of the system. To minimize the graphical view of the automaton, LTSA might label one arc with more than one action. These actions have an OR semantics, i.e., having *n* actions $a_i, \ldots, a_n$ labelling one arc is like having *n* arcs each one of them labelled with $a_i, \ldots, a_n$ respectively. The *"tau"* action means all the possible complementary actions with respect to the actions that are explicitly specified as performable from that node. By means of **L3** we specify - as valid behaviours of the system - all the ones in which the Backward component of the Master will always read from "e" only after that the Forward component of the Slave has read from "a". **L3** expresses a requirement of the correct functioning of the control loop. Satisfying **L3** assures that the information required to update the state of all the ControlComponent in the control systems loop is not available until all the Forward components have executed their code.

5.4 Analyzing Real-Time properties

In this section we will discuss the non-functional model of SaveCCM. We will show how we can analyze SaveCCM considering real-time properties in an automated way, and discuss a *resource reclaiming*-extension to SaveComp that utilizes spare capacities introduced by pessimistic real-time predictions. Further, we will also discuss analysis techniques and synthesis. In order to reason about real-time behaviour we need to transform the design-time components into tasks conforming to a real-time model. The tasks can then be analyzed considering the design requirements. The process is performed in the steps:

Model transformation: Model transformation involves the steps *(i) component to task allocation* and *(ii) attribute assignment* which are necessary in order to transit from the component model, to a run-time model enabling verification of temporal constraints and usage of efficient and deterministic execution environments.

Real-Time Analysis: To show that the run-time tasks will meet their stipulated timing constraints, schedulability analysis must be performed. We assume a fixed-priority systems (FPS) (the predominant scheduling method in today's real-time operating systems).

Synthesis: Synthesis involves mapping the tasks to operating system specific entities, mapping data connections to an OS specific communication, generating glue code, compiling, linking and bundling the program code.

5.4.1 Model transformation

When designing a control system with components, the designer is not required to consider the schedulability of the system, but should rather focus on the functionality. The components should be annotated with non-functional information corresponding to the control performance, e.g., periods and jitter constraints. Transformation of components to tasks and scheduling the tasks on a real-time operating are automated processes.

In order to reason about, e.g., real-time each component must be annotated with appropriate quality attributes. These quality attributes are; A finite worst-case execution time (**WCET**). A nominal period (**T**), and in the case it is appropriate jitter constraint (**Jitter**). The SaveCCM model also has transactions that can be used for defining timing constraints of data and/or control paths. Transactions has end-to-end deadlines (**E2ED**), that define the longest allowed latency between two components in the system.

The FPS model, which is used for analyzing the timeliness of the systems, defines a system as a set of tasks with a set of attributes. It is necessary to translate the components with their temporal constraints in to tasks The component to task converter performs two separate steps; firstly a transformation from components to task *(task allocation)*, and secondly *task attribute assignment*. To assign the FPS model attributes in such a way that the high level temporal constraints on transactions are met is non-trivial and has been addressed in research by e.g., [3], [16].

Attributes that are assigned during task attribute assignment are:

T (Period) - All periodic tasks have a period time that is assigned during the task allocation. Sporadic tasks have a MINT that analytically can be seen as a period time;

O (Offset) - The offset is an attribute that periodic tasks with jitter constraints are assigned. The earliest start time is derived by adding the offset to the period time.

P (Priority) - The priority is an attribute that indicates the importance of the task relative to other tasks in the system. In a FPS system tasks are scheduled according to their priority, the task with the highest priority is always executed first. All tasks in the system are assigned a priority.

WCET (execution time) - The worst case execution time is an attribute that is used for schedulability analysis and dynamic run-time scheduling, e.g., adaptive quality of service [6].

Components can be mapped to tasks in numerous ways. Common approaches are to map each component to one task, or all components to one single task. These two approaches may have obvious drawbacks, in the former, there may be extensive overhead in terms of, e.g., cpu-overhead (context-switches). In the latter, where all components are mapped to one task, the flexibility for the scheduler is lower and the timing requirements might not be fulfilled. Furthermore, when constructing systems the developer is often required to manually set task attributes such as priorities. Since the priorities directly decides how the tasks are scheduled, this is a hard task. In our approach this process is automated in the task attribute assignment plug-in.

[10] For the sake of simplicity we omitted the AIs and the AO.

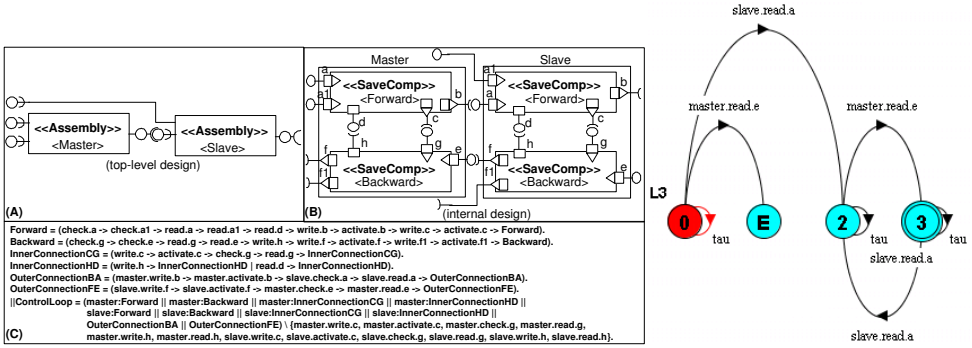

Figure 6. FSP Specification of a cascade control loop and an its liveness property

The context-switch time is increasing with the number of tasks, and the ideal mapping considering stack usage and task switch-overhead is to map all components to one task. However, in most cases this is not feasible due to the real-time constraints of the system. The task allocation strategy aims at increasing efficiency and dependability of the system since it is important to keep resources at a minimum in most embedded systems. It also strives at maintaining traceability and testability. Hence, the notion of components must be maintained both during development and after deployment. If the notion of components is lost, then traceability and testability are compromised.

A common approach to preserve the notion of components also after deployment is to use a one-to-one mapping between components and tasks, i.e., map one component to one task. However, the one-to-one mapping often implies worse resource usage than necessary. By using a stochastic state-space search technique we can, given certain criteria, find optimized mappings considering different properties, e.g., performance or dependability. By framing components during run-time by adding component-information on the stack (similar to a stack-frame), the notion of components is preserved when several components are mapped to one task.

In [7] a framework is proposed to facilitate the mapping between components and tasks by setting up mapping rules and exploit *Genetic Algorithms* (GA) to find feasible mappings that is optimized considering stack usage and context-switch overhead. This framework also constitutes the proposed plug-ins *Component to Task Converter*, *Task Attribute Assignment* and *Real-Time Analyzer*.

5.4.2 Real-Time Analysis

An important issue in obtaining high resource utilization is to deploy an efficient and tight schedulability analysis. The analysis need to faithfully model the complex execution behaviour that arises in control systems. Especially, the analysis should be able to handle arbitrary large jitter and deadlines , task synchronization and shared resources, and operating systems overhead.

For fixed-priority systems (the predominant scheduling method in today's real-time operating systems), the recent fast and tight response-time analysis (RTA) for tasks with offsets provides a suitable efficient and tight analysis [11]. This technique can model the precedence relations between tasks and hence gives a very accurate model of the system behaviour. Furthermore, the execution speed of this technique widely outperforms previous methods and is hence highly suitable for deployment in an optimization algorithm.

An efficient schedulability analysis requires an efficient prediction of WCET. Developers often use manual instrumentation methods in order to obtain WCET estimates. However, the accuracy is often low, hence to be safe the WCETs are often heavily overestimated. Current work on SaveCCM includes adding context-dependent and stochastic methods to predict WCET of SaveCCM components. [13, 12].

5.4.3 Synthesis

For synthesizing an assembly, platform specific API calls have to be inserted in the code. SaveCCM uses a general API and an API-translator (Code generator) The code generator resolves communication within and between tasks by translating platform-independent system calls with platform-specific system calls and adds platform specific glue code.

To maintain traceability and testability, it is important to maintain the notion of components, also after task allocation, code generation and deployment. Hence, we propose that the code generation module should be extended to add frames to components, similar to stack-frames. In other words, *enter component x* and *leave component x* can be pushed and pop on the stack. This leads, e.g., to easier debugging.

5.4.4 Resource Reclaiming Extension

Real-Time Analysis is based on worst-case behaviour in order to guarantee correct behaviour in all situations. Due to this, the analysis often becomes pessimistic because the worst-case scenario does not always reflect the actual case. Thus, when the worst case does not occur, there are left over resources in terms of processing time, i.e., residual time. The residual can be dynamically reclaimed and used for, e.g., dynamic property checking or other types of monitoring in low priority tasks.

The resource-reclaiming strategy is performed with an on-line service scheduler that uses hybrid scheduling to choose appropriate actions considering a residual time and. Low priority monitoring-tasks can use the residual time. The residual time can also be used for scheduling higher quality versions of the normal tasks as described in [6].

6 The ACC case study

We use an Adaptive Cruise Controller (ACC) prototype, implemented in SaveCCM [2], to evaluate our ideas (see Figure 7). The ACC extends the regular cruise controller in that it helps the driver to keep a safe distance to a preceding vehicle, autonomously changes the speed depending on the speed limit regulations, and helps the driver to slam the brake in extreme situations.

The application has three different operational modes: *Off*, *ACC Enabled*, and *Brake Assist*. In the *Off* mode, none of the control related functionality is activated. During the *ACC enabled*

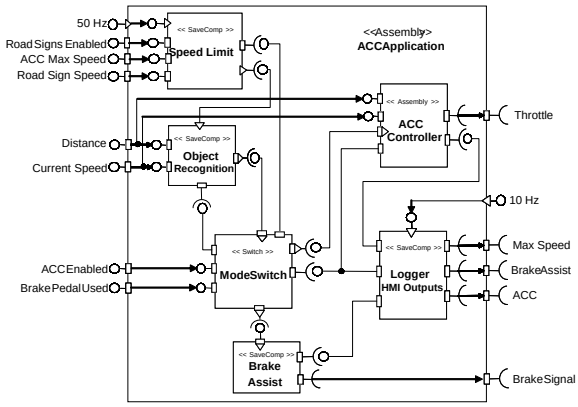

Figure 7. SaveCCM top-level design of the ACC application

mode the control related functionality is active. In the *Brake Assist* mode, braking support for extreme situations is enabled. The application (Figure 7) is based on four components, one switch, and one component assembly. To deliver the response for the time-critical computation as fast as possible, the assembly (ACC Controller, Figure 7) is, in turn, implemented as a cascade controller using two control modules as showed in Figure 8.(a). The two control modules Distance Controller and Speed Controller are two assemblies of ControlComponent type and they represent the master and the slave of the ACC Controller, respectively. They are connected through a connection among their Control ports (i.e., a ControlConnection). In Figure 8.(b) we show the internal design of ACC Controller as automatically derived by the *SaveCCM top-level Design Converter* plug-in in our framework. CalcOutput1 (CalcOutput2) and UpdateState1 (UpdateState2) represent the forward and the backward components of Distance Controller (Speed Controller), respectively.

Furthermore, the application has two different trigger frequencies, 10 Hz and 50 Hz. Logging and HMI output activities execute with the lower rate, and control related functionality at the higher rate.

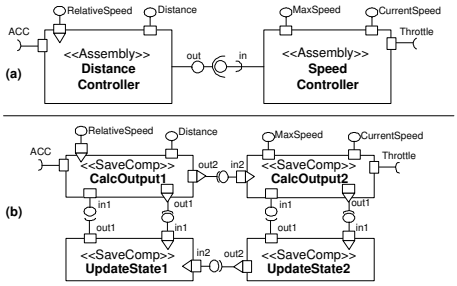

Figure 8. SaveCCM (a) top-level and (b) internal design of the ACC Controller assembly

For a detailed presentation of the ACC application functionality, we refer to [2].

6.1 Checking safety and liveness properties of the ACC application

We are interested in checking safety and liveness properties of the ACC application. On one hand, we want to check implicit (i.e., they do not have to be specified but can be directly observed by exploiting the model of the system) safety and liveness properties such as deadlock-freeness and livelock-freeness (i.e., progress). In other words, we are interested in checking that deadlocks do not occur and that every action can be eventually performed. Moreover, we want to also check specified safety properties such as **Safety1:** *"if the ACC Enabled (Brake Assist) mode is disabled then also the component ACC Controller (Brake Assist) must be disabled"* or liveness ones such as **Liveness1:** *"the information required to update the state of all the components in the ACC Controller is not available until all the output values have been calculated"*.

Validating the first property allows us to state that the ACC application is safe with respect to the different operational modes. Validating the second one allows us to state the correctness of the control loop performed by the ACC Controller with respect to updating its state based on the feedback signals. The assumption, here, is that each basic component in SAVEComp is a black-box one whose behaviour (in terms of the entry function that it performs) has been already validated with, e.g., unit testing.

For the purposes of our case study, we consider LTSA as *Safety and liveness analyzer*. Thus, we also consider FSP notation as specification language used by the developer to enrich the system design with safety and liveness properties that must be checked.

To check deadlock-freeness and progress, we only need to derive the model of the functional behaviour of the system in form of a FSP process (i.e., a LTS) without considering a property specification. We do this by exploiting the *Functional behaviours Models Generator*, which - analogously to what we have showed in Section 5.3 - derives the FSP model of the system's functional behaviour. This is done by taking into account the ACC application design as derived by the converter, the underlining execution model of the components forming it (as imposed by the SaveCCM semantics), the set of possible actions performable on the component ports and the possible values of the input ports ACC Enabled and Brake Pedal Used. These ports handle boolean values (i.e., they have a ON/OFF semantics) and they allow to enable the three different operational modes of the ACC application.

Once the generator has derived this model, the developer can easily interact with LTSA to verify that deadlocks do not occur or every action is eventually performed during the execution of the ACC application. For space reasons, here, we do not report the automatically derived specification of the ACC application[11].

Although LTSA exploits *partial order reduction* to efficiently perform the deadlock-freeness and liveness check, it suffers of the well known state explosion problem. It is worth noticing that, when building the model of the system requires too much memory, we can exploit the architectural constraints imposed by the pipe-and-filter style (SaveCCM is based on) to efficiently analyze the system by following a compositional reasoning. That is, we can analyze only parts of the system (i.e., its subsystems) by composing them with an *"efficient"* (with respect to memory consumption) environment that is semantically equivalent to the actual environment. In this way we obtain a minimized model of the system that is equivalent to the original one. This *"efficient"* environment can be automatically derived by taking into account the interface of the components forming the subsystem selected - by the developer - for the analysis. In fact, due to the execution model imposed by the pipe-and-filter style, it is the environment that provides the considered subsystem with the input and output data expected on its ports. After checking a functional property against the parallel composition of the considered subsystem with the *"efficient"* environment, the latter must be checked as functional property of the subsystem constituted by the actual environment. This is required to correctly carry on the

[11] It is available at the following URL: http://www.di.univaq.it/tivoli/ACCApplication.lts

analysis by following a compositional reasoning.

The previous compositional approach can be adopted to efficiently verify that both **Safety1** and **Liveness1** hold during the execution of the ACC Application.

To check **Safety1**, from the derived FSP specification of the system, the developer extracts the one of the subsystem formed by Mode Switch, ACC Controller and the connection among them. Then, by using the *models generator*, the developer can mechanically derive an *"efficient"* environment for the considered subsystem. This environment simply provides Mode Switch with the data expected on the input ports ACC Enabled and Break Pedal Used. It also provides ACC Controller with the data expected on the input ports Distance and Current Speed and gets the data sent on the output port Throttle. It is worth mentioning that, as showed in Section 5.3, the derived FSP specification contains also the model of the connection among the output and the input data-and-triggering ports (i.e., RelativeSpeed) of Mode Switch and ACC Controller, respectively. **Safety1** is specified in terms of the actions that can be performed on the ports ACC Enabled, Break Pedal Used and RelativeSpeed. For ACC Enabled and Break Pedal Used the reading actions have different names based on the possible operational modes. This is a possible way, in FSP, to exploit the boolean values of ACC Enabled and Break Pedal Used in order to model whether a specific operational mode is enabled or not.

To check **Liveness1** it is enough to extract only the FSP specification of ACC Controller, since this property is a *local* one and it is not related to the interaction with other components. Analyzing **Liveness1**, as property of ACC Controller, is done analogously to what we have done to analyze **L3** as property of the cascade control loop discussed in Section 5.3. Thus, for space reasons, we do not further discuss the analysis of **Liveness1**.

6.2 Real-time analysis of the ACC application

In this section we are interested in analyzing the real-time behaviour of the application and prove that the constraints imposed on the system will hold. The 'ACC Application' constitutes 9 components. All components except one is triggered by a 50 Hz external periodic trigger. The last component is triggered by an 10 Hz external periodic trigger.

For the simplicity we assume that each component has a WCET of 1 ms. We assume that the context-switch time is 0.1 ms, and we also constrain the system with four transactions, denoted $tr_i < (components); E2ED >$

tr_1 *<(Speed unit,Object rec.,Mode Switch,ACC Con.);10 ms>*
tr_2 *<(Speed unit, Mode Switch, ACC Con.);10 ms>*
tr_3 *<(Speed unit,Object rec.,Mode Switch,Break Assist);5 ms>*
tr_4 *<(Logger Unit);20 ms>*

We will briefly demonstrate how the system is transformed from components to tasks and the resulting task properties derived from the components. The process is tools oriented and the detailed workflow of each tool is not described due to space limitations.

The components and transactions are transformed into FPS tasks by applying the model-transformation, response-time analysis (RTA) and synthesized iteratively, giving feedback to the designer considering fulfilled constraints, and/or possible failures.

The *component to task converter* plug-in searches for an optimized mapping from components and tasks considering context-switch overhead, stack size and the stipulated real-time constraints, i.e., the end-to-end deadlines of the transactions. The *task attribute assignment* plug-in assigns the tasks the derived attributes period, offset, priority and WCET. The *real-time analyzer* plug-in performs response-time analysis to ensure that the constraints are met.

In this example, with the above stated constraints, the system is divided into three tasks as depicted in figure 9. Task A maps the components *Speed Unit, Object Recognition, Mode Switch* and *Break Assist*. Task B maps the assembly *'ACC controller'* and task C maps the *logger* component. The derived attributes assigned to the resulting tasks are:

Task A: Period 20 ms, Offset 0, Priority High, WCET:4 ms
Task B: Period 20 ms, Offset 0, Priority Mid, WCET: 4 ms
Task c: Period 100 ms, Offset 0, Priority Low, WCET: 1 ms

The response-time analysis shows that the transactions $tr_1 - tr_4$ will have the worst case response-times, 8.1 ms, 8.1 ms, 4 ms and 9.2 ms respectively. By comparing these values to the end-to-end deadline (E2ED) constraints we can see that all transactions will be met and the system is correct considering timing.

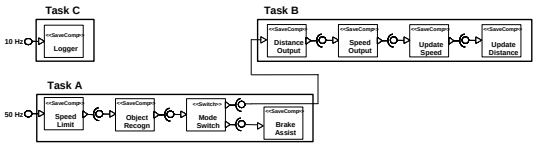

Figure 9. Resulting task set after model transformation

Considering the systems different modes, we see that some modes will not require all components to be run, leaving some of the tasks with a significantly shorter execution-time than WCET. In this case an on-line scheduler can use the residual (left over) time to schedule either low priority monitoring tasks or higher quality versions of the standard tasks. In both cases increasing the quality of the system in some way.

7 Conclusion and future work

Although component models that support predictability of the system behaviour there exist, they are found to be inappropriate for the control systems application domain since they do not support the requirements of embedded systems and, hence, are not able to predict the behaviour of control systems. The approach presented in this paper represents a possible solution to this problem. By means of it, we can build/compose control systems components (i.e., in designing the control system we can use a component-based approach by exploiting all its notorious advantages) and - in the same time - predict the functional/non-functional behaviour of the composed system. Although extending SaveCCM with the possibility to specify a top-level design of the system considerably simplify the developer tasks, it internally adds complexity at level of system implementation. To validate the real feasibility of our approach, as future work, we plan to apply SAVEComp to real-scale case studies. Moreover, SAVEComp, as it is currently structured, still lacks of integration between functional and non-functional analysis. That is, functional and non-functional analysis are separately performed. We also plan to incorporate SAVEComp into TOOL•ONE framework [5] which supports functional and non-functional analysis integration, and implement the SAVEComp parts that go beyond the approach presented in this paper.

Acknowledgements

This work is supported by SSF within both SAVE (*SAfety critical components for VEhicular systems - http://www.mrtc.mdh.se/SAVE/*) and FLEXCON (*FLEXible embedded CONtrol systems - http://www.control.lth.se/FLEXCON/*) project.

8 References

[1] *International Electrotechnical Commission, IEC 61131 Programmable Controllers. Part 1 - 5*, January 1992.

[2] M. Akerholm, A. Möller, H. Hansson, and M. Nolin. Towards a Dependable Component Technology for Embedded System Applications. In *Proceedings of the 10th IEEE International Workshop on Object-oriented Real-Time Dependable Systems (WORDS05)*, February 2005. Sedona, Arizona, USA.

[3] A. Bate and I. Burns. An approach to task attribute assignment for uniprocessor systems. In *Proceedings of the 26th Annual International Computer Software and Applications Conference*. IEEE, 2002.

[4] J. Buchi. On a decision method in restricted second order arithmetic. In *International Congress on Logic, Method and Philosophical Sciences*, 1960.

[5] V. Cortellessa, A. Marco, P. Inverardi, F. Macinelli, and P. Pelliccione. A framework for the integration of functional and non-functional analysis of software architectures. In *TACoS*, 2004.

[6] J. Fredriksson, M. Akerholm, K. Sandström, and R. Dobrin. Attaining flexible real-time systems by bringing together component technologies and real-time systems theory. In *Proceedings of the 29th Euromicro Conference, Component Based Software Engineering Track*, September 2003. Belek, Turkey.

[7] J. Fredriksson, K. Sandström, and M. Akerholm. Calculating resource trad-offs when mapping components to real-time tasks. In *In the 8th International Symposium on Component-Based Software Engineering (CBSE8), St.Louis, USA*, May 2005.

[8] H. Hansson, M. Akerholm, I. Crnkovic, and M. Törngren. SaveCCM - a Component Model for Safety-Critical Real-Time Systems. In *Proceedings of 30th Euromicro Conference, Special Session Component Models for Dependable Systems*, September 2004.

[9] B. Lewis. IEC 61499 Function Blocks: A new way to design control systems? *Control Engineering Europe*, April 2002.

[10] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. John Wiley and Sons, 1999.

[11] J. Mäki-Turja and M. Nolin. Fast and Tight Response-Times for Tasks with Offsets. In *17th EUROMICRO Conference on Real-Time Systems*. IEEE, July 2005. Accepted for publication.

[12] A. Möller, J. Fredriksson, I. Peak, M. Nolin, and H. Schmidt. Context Dependent Predictions of Component-Based Control Software. In *Submitted to ERCIM Workshop on Dependable Software Intensive Embedded systems*. IEEE Computer Society, Septembed 2005.

[13] T. Nolte, A. Möller, and M. Nolin. Using Components to Facilitate Stochastic Schedulability. In *Proceedings of the 24th Real-Time System Symposium – Work-in-Progress Session*. IEEE Computer Society, December 2003. Cancun, Mexico.

[14] E. Parr. *Programmable Controllers - An Engineer's Guide (2nd Edition)*. Butterworth-Heinemann Ltd, 2001.

[15] L. Pernebo and B. Hansson. Plug and play in control loop design. In *Preprints Reglermöte 2002*, Linköping, Sweden, May 2002.

[16] C. Sandström, K. and Norström. Managing complex temporal requirements in real-time control systems. In *In 9th IEEE Conference on Engineering of Computer-Based Systems Sweden*. IEEE, April 2002.

[17] K. Sandström, J. Fredriksson, and M. Akerholm. Introducing a component technology for safety critical embedded real-time systems. In *Springer - LNCS 3054*, May 2004.

[18] M. Tivoli, J. Fredriksson, and I. Crnkovic. A component-based approach for supporting functional and non-functional analysis in control loop design. *Malardalen University, Malardalen Real-Time Research Centre. Technical Report*, May 2005.