A component-based approach for supporting functional and non-functional analysis in control loop design

Massimo Tivoli[1], Johan Fredriksson[2] and Ivica Crnkovic[2]
[1]University of L'Aquila, Computer Science Department, Italy, tivoli@di.univaq.it
[2]Mälardalen University, Mälardalen Real-Time Research Centre, Västerås - Sweden
{jhoan.fredriksson, ivica.crnkovic}@mdh.se

Abstract

One of the main issues in developing modern control systems is how to design control loops in such a way that functional requirements as well as real-time attributes can be analyzed during design-time. Nowadays, control loops are often constructed performing a modular approach by means of libraries of building blocks that can be considered as components of a control system. Although component models that support predictability of the system behavior there exist, they are found to be inappropriate for the control systems application domain. In fact, they assume to not deal with control loops (i.e., control flow feedbacks) which cause problems with predictability of the system behavior. This has lead to a real need of a component-based approach for designing control loops and supporting predictability of the behavior of the designed system. In this paper, we present a possible component-based approach for supporting functional and non-functional analysis of control loops during design-time. Moreover, we outline an overall view of the component-based development framework which implements our approach.

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. Recently, 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). One of the main issues in control engineering domain is how to design control loops [11] 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 control loop life-cycle, namely during design-time. Nowadays, control loops 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 loops out of a set of *"control modules"* [12]. Such 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 as a component of a control system and hence it is the mean to build control loops by adopting a component-based approach supported by a suitable component technology. Unfortunately, commercial component technologies are too complex and unpredictable and hence, within such an approach, predictability of the functional and non-functional behavior 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 behavior there exist, they assume to not deal with control loops (i.e., control flow feedbacks) which notoriously cause problems with predictability. Thus, a component-based development framework which supports predictability of the functional and non-functional behavior of a control system, during design-time, is highly needed.

In this paper, we present a possible component-based approach supporting predictability of the control system behavior during control loops design-time. Our reference component model is *"SaveCCM"* [6] which is designed for safety-critical real-time systems. SaveCCM is part of a component-based development framework called SAVEComp. SAVEComp is being developed in the project SAVE (*SAfety critical components for VEhicular systems - http://www.mrtc.mdh.se/SAVE*). The main purpose of SAVEComp is to provide efficient support for designing and implementing

[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 [7]).

embedded control applications by mainly focusing on simplicity and analysability of functional requirements and of real-time and dependability quality attributes. SaveCCM has been thought to support predictability of the real-time behavior of the system. 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 behavior. A control module in SAVEComp is inherently able to correctly deal with outer and inner control loops. 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 on the loop that are needed for the correct functioning of the loop and to predict the system behavior. 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, the developer is able to build and compose control modules in such a way that both functional requirements and real-time attributes can be analyzed in control loops design.

The paper is organized as follows. Section 2 sets the context of our work by referring to control modules as a solution for an "easy-to-make" component-based design of control loops. In Section 3 the main features of SaveCCM are summarized. In Section 4 we first outline the overall structure of SAVEComp and then - by means of an explanatory example - we present our approach as it is implemented within SAVEComp. Section 5 concludes and discusses future work.

2 Setting the context

In Section 1, we said that in many modern control systems, a control loop is designed by using a modular approach in which its constituent function blocks are combined together. This is the case, e.g., of cascade control loops [11]. Unfortunately, nowadays, 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 designer should have a deep knowledge of the different function blocks. The second one is the obvious risk to make mistakes when the designer has to deal with a large amount of parameters and interface points. In [12], a component-based solution to overcame these disadvantages has been proposed. The main idea is to reduce the complexity of control loops 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 will 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 [12]. This data structure contains all the signals that are sent between the function blocks of the control loop. Since in real-scale control loops some of the signals are sent forwards and some are sent backwards, **ControlConnection** collects all the signal in two substructures called **Forward** and **Backward** respectively.

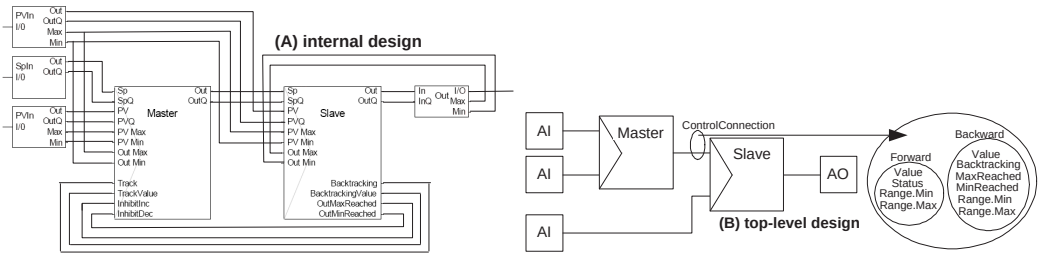


Figure 1: Two different designs of the same cascade control loop

In Figure 1.**A** we show an example of a cascade control loop 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 graphical connection of **ControlConnection** type. Note that a control loop is configured in a much simpler way if the blocks are connected with a **ControlConnection** structure by considerably simplifying the designer tasks. 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, within the loop, without delays. For this reason, in [12], 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 without delay, respectively. Although the control module concept considerably reduces the complexity of control loops by providing a component-based approach to design them, unfortunately current component technologies do not allow one to realize a control module in order to provide the designer with facilities for supporting predictability of the control system behavior. 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 behavior.

3 The SaveCCM component model

In this section we briefly describe the main characteristics of our reference component model called SaveCCM. Refer to [6] for a detailed description of it. For a comparison of SaveCCM to other component models, please refer to [5]. SaveCCM consists of the following main architectural elements: (1) **components**, which are basic units of encapsulated behavior; (2) **switches**, which provide facilities to statically and dynamically change the component interconnection structure; (3) **assemblies**, which provide means to form aggregate components from sets of interconnected components and (possibly) switches; and (4) **run-time** framework, which provides a set of services, such as communication between components, component execution and control of sensors and actuators. The functional interface of all architectural elements is defined in terms of a set of ports, which are points of interaction between the element and its external environment. We distinguish between input and output ports, and there are two complementary aspects of ports: the data that can be transferred via the port and the triggering of component executions. SaveCCM 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 the component triggers have been all activated, the component reads its input ports (i.e., *reading* state), performs its computations (i.e., *executing* state) based on the inputs read and its internal state, writes the result of the execution on its output ports (i.e., *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.

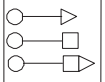
Symbol	Interpretation	Symbol	Interpretation
	**Input ports** - The upper symbol is an input port with a trigger, and no data. The middle symbol is an input port with data and no triggering, and the lower one is an input port with data and triggering.	<<SaveComp>> **<name>**	**Component** - A component 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

A component behavior is defined by means of variables that express internal states, and actions that describe the component execution. Variables can be initiated by values from the input ports. Actions are abstract specifications of the externally visible behavior of the component. Components are specified by their interfaces, behavior and (quality) attributes. A subset of the UML2 component

diagrams[2] is adopted as graphical specification language[3]. The symbols showed in Figure 2 are used.

4 The SAVEComp development framework

In this section we outline the overall structure of the SAVEComp development framework (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 safety-critical embedded real-time systems. A part of it is the AutoComp technology [13] which is intended only for predicting the real-time behavior of the system.

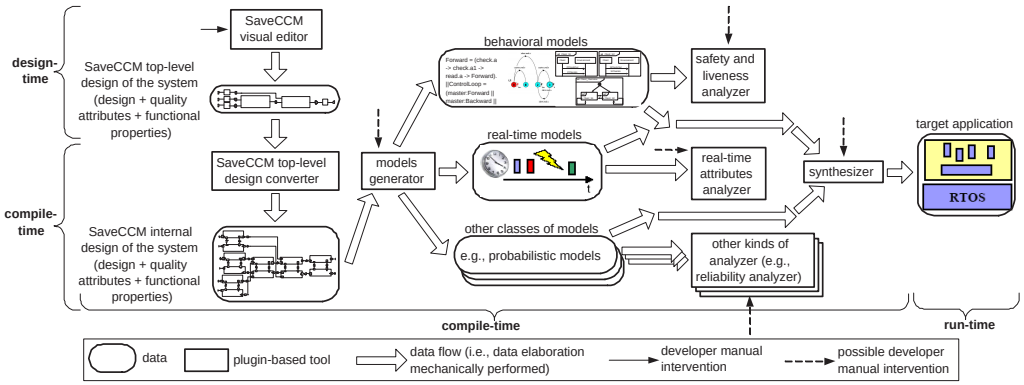


Figure 3: The SAVEComp development framework

As showed in Figure 3, SAVEComp can be described by distinguishing three main phases of its utilization. During design-time, developers can 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 (i.e., by using control modules). Moreover, the extended version of SaveCCM allows the developer to enrich the top-level design with: (1) functional properties of the system expressed in terms of sequences of actions performed on component ports; and (2) 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 one and, from it, 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) behavioral models (e.g., Process Algebras, LTSs, state machines, MSCs, UML2 interaction diagrams); and (ii) real-time models (e.g., Fixed Priority Scheduling models). The first class is intended to perform functional analysis (i.e., checking safety and liveness properties), the second one to perform non-functional analysis in the specific case of guaranteeing real-time attributes. The plugin-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 plugin-based tool within SAVEComp. Each "plugin" 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 some functional or non-functional requirement has not been met or - if the design matches every requirement - execute a synthesis step. It is within this step the binary representation of the system is created, often the operating system and the run-time one are also included with the application code in a single bundle. Moreover, in each utilization phase, the developer has the possibility to interact with a particular plugin-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 chosen *Eclipse* platform[4] as implementation environment of SAVEComp since it provides

[2]UML2.0 specification - http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML.
[3]In [6], the complete textual syntax (i.e, BNF specification) of the specification language is reported.
[4]The Eclipse project. Eclipse platform technical overview. Technical report, 2001 - http://www.eclipse.org.

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. This XML core is the 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 our approach.

4.1 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 the loop) without delays, 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. 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.

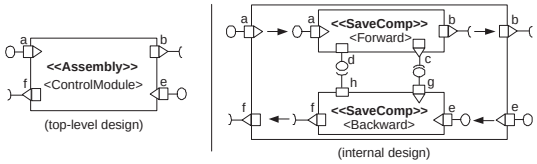


Figure 4: Top-level and internal design of "ControlModule"

It is worth mentioning that Forward and Backward, as usual SaveCCM components, respect the component execution model mentioned in Section 3. 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[5] (i.e., for modeling 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. Forward and Backward handle the Forward and Backward substructure, respectively. The triggering data are the mean which is used to activate a Forward component or a Backward one depending on the control flow of the loop. They assure that the information required to update the state of all the ControlComponent 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 loop. Note that a ControlComponent can handle outer control loops as well as inner ones. 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.

4.2 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.

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

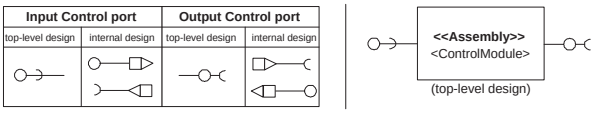


Figure 5: Top-level and internal design of a Control port and final top-level design of "ControlModule"

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.

4.3 Analyzing functional requirements

In this section we formalize the execution model of a ControlComponent. This formalization is intended to support functional analysis of control loops during design-time. We are interested in proving safety and liveness properties. To formalize the execution model of a ControlComponent we have to look at (i) its internal design; (ii) the execution model of a SaveCCM component; and (iii) the set of possible actions on a SaveCCM port. By referring to Section 3, 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 behavior of a component, with an arbitrary number of ports of different type[6], by means of a process algebra. We choose FSP [8] (*Finite State Processes*) as process algebra to model the execution behavior 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 [8] (*Labeled Transition System Analyser*). LTSA is a plugin-based verification tool for concurrent systems. It mechanically checks that the specification of a concurrent system satisfies required properties of its behavior. In addition, LTSA supports simulation to facilitate the interactive exploration of the system behavior. 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 loop - showed in Figure 1 - as specified by the designer using the *"SaveCCM visual editor"*[7], 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 *"models 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 *"models generator"*. Integrating SAVEComp with LTSA (i.e., a possible *"safety and liveness analyzer"*) allow us to easily verify functional properties of the loop's FSP specification. For example, we can mechanically verify that deadlocks do not occur in the execution of the control loop (i.e., safety). Moreover, we can also verify that the execution of the

[6]Note only that if a component C - beyond other ports - has also $p_1, ..., p_n$ input data-and-triggering ports then - during the initial part of its execution - C will execute a sequence of n "check" (each one of them for each p_i) followed by a sequence of n "read".

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

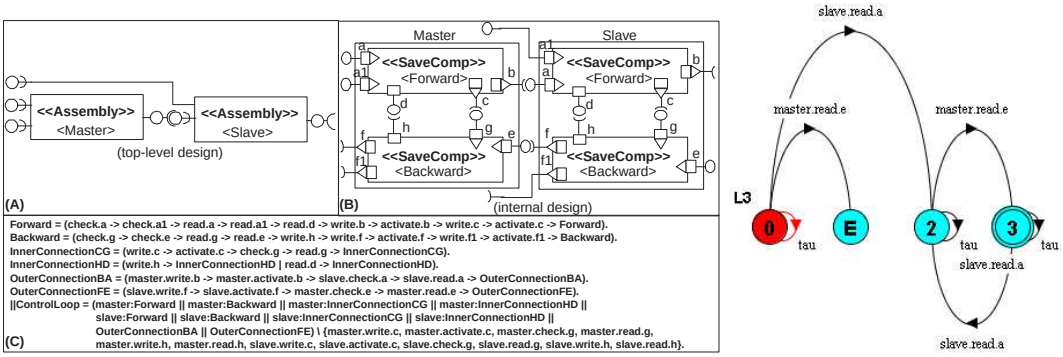


Figure 6: FSP Specification of the cascade control loop and an its liveness property

control loop 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 [3]. Informally, the Büchi Automaton is an operational description of the property and specifies the set of system behaviors 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[8]. By means of **L3** we specify - as valid behaviors of the loop - all the ones in which the Backward component of the Master will always read form "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 loop is not available until all the Forward components have executed their code.

4.4 Guaranteeing real-time requirements

In this section we will discuss non-functional requirements (NFRs), real-time theory and transformation from design to run-time models. Period, end-to-end deadline, jitter and WCET are required to reason about guaranteeing real-time requirements. Periods, end-to-end deadlines and jitters are NFRs that are imposed by the environment. The WCET is a non-functional property (NFP) that depends on the implementation and the underlying run-time system and varies between different platforms. 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. [10, 9]. By guaranteeing real-time requirements we mean to find a feasible execution schedule that fulfills the given NFRs of the ControlComponents, i.e., periods, end-to-end deadline and jitter constraints. In order to calculate if the specified NFRs can be fulfilled, real-time analysis (RTA) with transactions is utilized as described in, e.g., [14]. However, to use RTA the ControlComponents must be organized into schedulable entities (tasks). Components can be mapped to tasks in numerous ways and 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 memory stacks and control blocks, and 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, the designer is often required to manually set task attributes such as priorities. An important issue in embedded systems is to keep resources at a minimum while guaranteeing predictability. In [4] a framework is developed that utilize stochastic search methods to allocate components to tasks and derive task-attributes such as periods and priorities in such a way that real-time properties are guaranteed (if possible) and overhead in terms of cpu-overhead and memory usage is minimized. Consider the Figure 6.**A** and lets assume that the master and slave have requirements that force them to execute periodically at 10 Hz. Lets further assume that they have a 20 ms end-to-end deadline from input to the output. During compile-time, the master and slave are unfolded to their basic components as shown in Figure 6.**B** and their WCETs are determined. To simplify we assume that all four basic components have WCETs of 10

[8]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 $a_i, ..., a_n$ labeling one arc is like having n arcs each one of them labeled with $a_i, ..., a_n$ respectively. The *"tau"* action means all the possible complementary actions with respect to the actions that are explicitly specified as performable from that node.

ms. The basic components must be mapped to tasks in such a way that (i) Master and Slave Forward components fulfil the end-to-end deadline of 20 ms; and (ii) all components execute within 100 ms (one period). The best mapping in this specific case, considering real-time and low resources, is to map *Master.Forward, Slave.Forward, Slave.Backward, Master.Backward* to one task in that specific order.

5 Conclusion and future work

Although component models that support predictability of the system behavior there exist, they are found to be inappropriate for the control systems application domain since they are not able to predict the behavior of control loops. The approach presented in this paper represents a possible solution to this problem. By means of it, we can build/compose control modules (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 behavior of control loops. Although the introduction of the control module concept in SaveCCM considerably simplify the designer 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 [2] 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. We acknowledge Mikael Åkerholm which has provided the authors with very valuable comments and constructive suggestions to improve the paper.

References

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

[2] 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.

[3] E. M. Clarke and O. Grumberg and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[4] J. Fredriksson, K. Sandström, and M. Åkerholm. 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.

[5] J. Fredriksson, M. Tivoli, and I. Crnkovic. A component-based development framework for supporting functional and non-functional analysis in control systems design. Technical report, Technical report, Department of Computer Scienc and Electronics, Mälardalen University, 2005. Submitted for publication.

[6] H. Hansson, M. Åkerholm, I. Crnkovic, and M. Törngren. SaveCCM - a component model for safety-critical real-time systems. In *Euromicro Conference, Special Session CMDS*. IEEE, 2004.

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

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

[9] A. Möller, J. Fredriksson, I. Peak, M. Nolin, and H. Schmidt. Context dependent predictions of component-based control software. Technical report, Technical report, Department of Computer Scienc and Electronics, Mälardalen University, 2005. Submitted for publication.

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

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

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

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

[14] K. Tindell. Adding time offsets to schedulability analysis. Technical report, Department of Computer Science, University of York, 1994.