Synthesizing Real-Time Components to Run-Time Tasks

by Kathrin Dannmann
C. v. O. University of Oldenburg
kathrin@dannmann.de
February 11, 2009

Diploma Thesis

First Examiner:	*Prof. Dr. Martin Fränzle, University of Oldenburg*
Second Examiner:	*Dr. Thomas Nolte, Mälardalen University*

conducted at:
Mälardalen Real-Time Reseach Centre
School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden

Abstract

Component-based software engineering (CBSE) has proven to be effective for the development of desktop applications. Its objective is to construct software applications from reusable entities. The possibility to manage the increasing complexity, the reduction of time-to-market and the decrease of maintenance efforts can be mentioned as advantages brought by CBSE.

The SAVE and PROGRESS projects' objective is to show the potential of CBSE also for the embedded domain. This domain imposes a series of extra-functional requirements upon the development process, typically in respect to timing, resource usage and dependability. A component technology for embedded software should provide respective support, e.g., by means of analysis tools.

An important step during the development of embedded software applications with CBSE is the synthesis, i.e., the transformation of the component-based designs into run-time entities. The result of the synthesis affects a number of system properties, thus it is beneficial to use a deployment mechanism which is more elaborated than a plain code generation step. A high number of tasks, e.g., causes a high amount of context switches. Putting all components in one task on the other hand can result in a high processor load. In this thesis, a functional approach for the mapping of components to tasks will be presented.

The allocation mechanism is based on a control flow analysis and a set of rules, according to which the components are allocated to a number of tasks. The allocation is deterministic, which provides for predictability and allows for further analysis. The result of the components-to-tasks allocation is used for the concluding code generation step, in which task source code files in C are automatically generated.

The applicability of the designed synthesis mechanism is presented by use in a demonstrator project, which aims at showing the capability of the SAVE component technology.

Contents

Chapter 1

Introduction

1.1 Introduction

Component-based software engineering (CBSE) has shown to be an effective approach for, e.g., the development of desktop applications. It addresses important issues in the software development process, as for example the possibility to manage the increasing complexity, reducing time-to-market and the decrease of maintenance efforts, by creating systems from existing components.

Recent research projects aim at demonstrating the potential of CBSE also for the embedded domain. This domain imposes special requirements on the software that need to be considered during the development, e.g., real-time requirements, dependability and resource consumption. Potential benefits of applying CBSE for the embedded domain are, amongst others, shorter development time and support for product-lines due to the option for reuse, as well as decreased effort for upgrading and maintenance.

A component model and the corresponding component technology for embedded software design has been developed in the SAVE research project[1]. The component technology developed within the SAVE project has been designed to support component-based development of vehicular software. It includes mechanisms that pay regard to the special requirements of the embedded domain during the whole software design process. The PROGRESS project[2] will supersede SAVE with a broadened scope and build on the results that have been obtained.

This thesis has been carried out within SAVE and PROGRESS. Its objective is the deployment of the component-based designs for the execution on a target system. The requirements of the embedded domain should also be considered during this synthesis step, because it affects several properties of the system. The main task that needs to be performed during the synthesis is the allocation of components to task. The approach

[1]see http://www.mrtc.mdh.se/SAVE/
[2]see http://www.mrtc.mdh.se/progress/

which will be presented in this thesis has been based on a control flow analysis.

1.2 Related Work

Component technologies generally provide a deployment mechanism for the automatic generation of source code from higher level system designs. However, the allocation of components to tasks is often not considered.

[Stewart et al., 1997] describes a component concept called port-based objects (PBOs), which in combination with specific real-time operating system (RTOS) mechanisms builds a framework that supports the design and implementation of sensor-based control systems. The framework provides an automatic code generation from a PBO design to C, which minimizes the manual coding effort for the system developer. One run-time task is created from every PBO, i.e., a one-to-one allocation.

The COMDES-II framework, presented in [Ke et al., 2007], also supports the automatic generation of C-code from the higher-level component-based designs. The allocation of components to tasks is not explicitly mentioned.

The PECOS framework ([Genßler et al., 2002]) pursues a different strategy. The code generation takes advantage of the concepts of object-oriented programming languages. The component-based designs are mapped into a class hierarchy in either Java or C++. A detailed description is available in [Schönhage and van den Born, 2002]. However, the mapping of components to tasks is not mentioned.

The Rubus component model takes the mapping of components to tasks into account. According to [Hänninen et al., 2008], "*the RubusCMv3 model provides possibilities to map several components into one executable thread, minimizing contexts switch overhead.*" However, a detailed description of the allocation mechanism is not given.

An approach to map components to run-time tasks is presented in [Fredriksson et al., 2005]. It uses simulated annealing and genetic algorithms to find an optimal allocation, concerning a specified system property. For the work in this thesis, probabilistic or heuristic methods are not suitable, since the allocation strategy is aspired to be predictable and, e.g., applicable for hard real-time analysis.

1.3 Thesis Goal

The goal of this thesis is to design and implement an approach for the automatic deployment of SaveCCM applications (see 2.2), i.e., the translation of component-based software design into an executable format. The main objective is the mapping of components to tasks, which is to be done in a deterministic manner to provide for predictability.

The component-based designs are delivered by means of system descriptions in an XML format. The expected final result of the synthesis is task source code in C which implements the designed system. The source code will be compiled for a target system with an available standard compiler.

This report is organized as follows.

Chapter 2

The background on which this thesis is based is presented. This chapter particularly includes the introduction of the SaveComp component model.

Chapter 3

First, the approach which has been chosen for the component to task allocation will be introduced and motivated, whereupon the implementation of the designed synthesis mechanism is presented. This chapter covers the essential part of the work which has been carried out within this thesis.

Chapter 4

The testing of the implemented synthesis tool is described and the results, which have been achieved within this thesis, are presented.

Chapter 5

The work and achievements which have been accomplished within this thesis are summarized. An outlook on potential further development on the presented synthesis conclude the thesis report.

Chapter 2

Theoretical Background

2.1 Component-Based Software Engineering

Component-Based Software Engineering (CBSE) is a comparatively young branch in the software engineering discipline. Its objective is to construct software applications from reusable entities, i.e., software components, following the common practice in mechanical engineering where constructions are mainly assembled from already existing parts. By modifying or replacing single components in the application, maintenance and upgrading can be facilitated [Crnkovic, 2001]. CBSE comprises the development of applications from components as well as the development of reusable components, the building blocks for applications.

With this strategy, the development time for software systems can be reduced, due to the possibility for reuse, and therefore lead to a shorter time-to-market. Moreover, maintenance efforts can be decreased. CBSE has shown effectiveness in development of desktop and web applications [Lüders, 2004].

2.1.1 CBSE for Embedded Systems

Software development for embedded systems becomes more and more challenging. While the software is expected provide increasingly sophisticated functionality, the time to market decreases at the same time. The costs for embedded software development are rising continually as the software developers struggle to meet the tight deadlines [Kang et al., 2005].

These problems can be addressed by applying CBSE for embedded software. CBSE has shown to be an effective way to handle complex software designs and also to shorten the development time. The development of an application does not need to start from scratch, but can build upon previously achieved results. Furthermore, CBSE can provide support for product lines, which are common in many embedded domains, e.g., the mobile phone or car industry, since it allows for reuse and reduces the effort for maintenance and upgrading.

The embedded domain imposes special requirements on the software and thereby also on the development process. Embedded computers are often responsible for safety critical control tasks, which implies that they have to fulfill real-time requirements. Moreover they are produced in a high quantity and the amount of physical space in the system into which the computer shall be integrated is limited. To keep the costs for the utilized hardware within the bounds of economic possibility, the cheapest feasible option will be chosen which leads to tight resource constraints. Resource-efficiency, predictability and safety are important issues that should be considered throughout the whole software life-cycle. This implies that a component technology suitable for the software development for embedded systems should provide opportunities for analysis and verification in all stages of the development [Åkerholm et al., 2005].

Furthermore the constraints should be regarded during the deployment of the software designs, i.e., the transfer of components into run-time entities. The method how components are mapped to tasks has impact on for instance memory consumption, processor utilization and schedulability. A low number of tasks, for example, consumes less memory and involves less context switches, yet it can cause a high processor utilization when the components have different periodicities [Fredriksson, 2008].

2.2 The SaveComp Component Model

This thesis has been carried out within the SAVE[1] and PROGRESS research project. A core part of the SAVE project is the SaveComp component technology, herein after referred to as SaveCCT, which has been designed to support component-based development of vehicular software. The SaveCCT incorporates a series of tools that provide support for the special requirements of the embedded domain throughout the whole software design process.

The underlying component model is the SaveComp component model, in the following referred to as SaveCCM, which provides the syntax and semantics of the component language.

The remainder of this section will introduce the elements of SaveCCM. The graphical representation of the elements is presented in Figure 2.1. Further details about SaveCCM can be found in [Carlson et al., 2006] and [Åkerholm et al., 2007b]. [Åkerholm et al., 2007a] delivers insight into the SaveComp component technology.

[1]See http://www.mrtc.mdh.se/SAVE/ for further information.

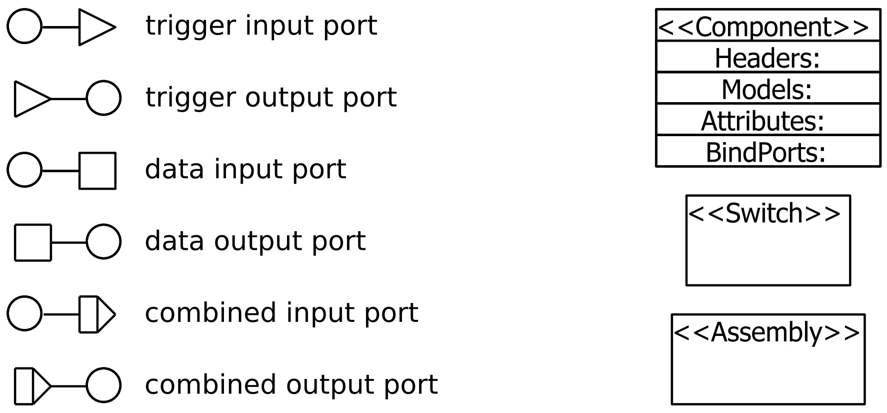


Figure 2.1: The graphical representation of the SaveCCM elements

2.2.1 Components

Components are the main element in SaveCCM. A component's interface consists of input and output ports and can also include a number of quality attributes, such as (worst case) execution time, reliability estimates or safety models. The functionality of a component is typically implemented by an entry function in C.

The execution of a component follows strict "read-execute-write" semantics. Initially, the component is inactive and remains in that state until it is *triggered*, i.e., all input trigger ports have been activated. The execution starts with a *read* phase, in which the data from the input ports is transferred for calculations during the subsequent *execute* phase. Once the calculations have been performed, the execution will conclude with a *write* phase. The output data is written to the output data ports, the output trigger ports are activated and the component transits into the inactive state again.

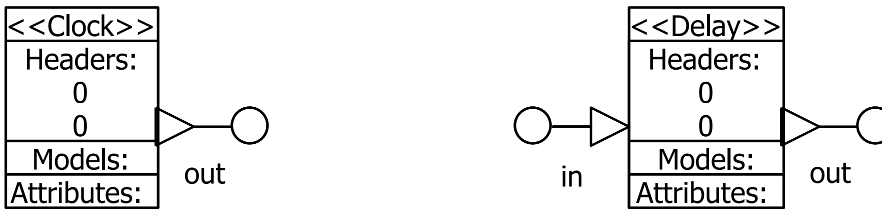


Figure 2.2: Graphical representation of the Clock and the Delay component

The model also includes two special types of components, *Clock* and *Delay* (see Figure 2.2), which can be used to manipulate timing of triggers. A *Clock* has two parameters T and J for period and jitter. A trigger is generated within J time units after the start of a new period, every T time units. A *Delay* component has two parameters D and P, for delay and precision, and will pass on an incoming trigger between D and $D + P$ time units after its arrival.

2.2.2 Assemblies

Assemblies are used to encapsulate a set of components. An example assembly can be seen in Figure 2.3. They contain an internal structure of components and interconnections, which is by this means hidden from the rest of the system and can only be accessed through the assembly's ports.

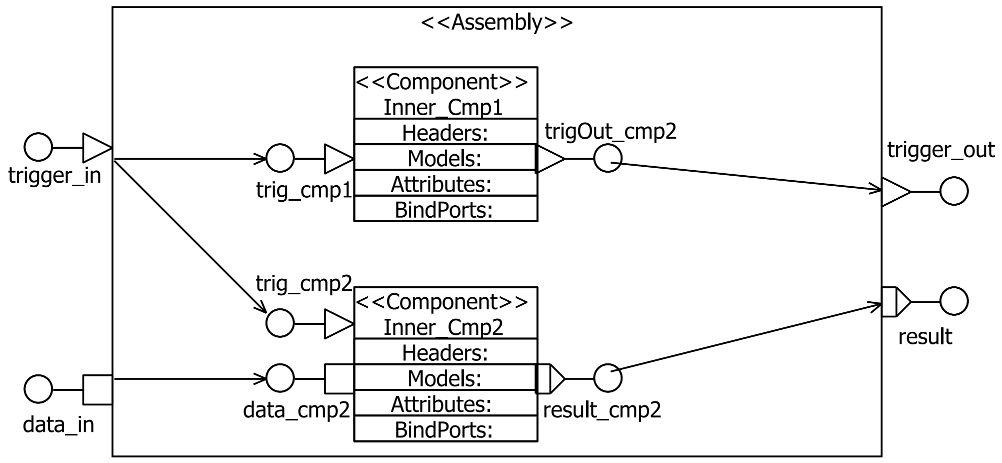


Figure 2.3: An example for an assembly

Assemblies do not fulfill the requirements of a component. They are not triggered, but immediately pass incoming data and trigger signals according to their internal connection. Therefore, assemblies can solely be considered a means to assemble and name a group of components, in contrast to Composite Components (see Section 2.2.3), which comply with the execution semantics of a SaveCCM component.

2.2.3 Composite Components

A composite component is a component whose behavior is specified by an internal composition of components. The graphical notation is shown in Figure 2.4.

The dashed lines indicate how data is transferred to and from the internal composition during the read and write phase, respectively. Trigger signals are not passed on in this manner. The internal trigger ports will be activated once the composite component enters the active state, i.e., when all its trigger ports have been activated. When all internal components become inactive again, the write phase will be initiated and the composite component will return to the idle state again.

2.2.4 Switches

Switches allow to modify the component interconnection structure. The modifications can either be conducted statically for pre-runtime static configuration or dynamically, for

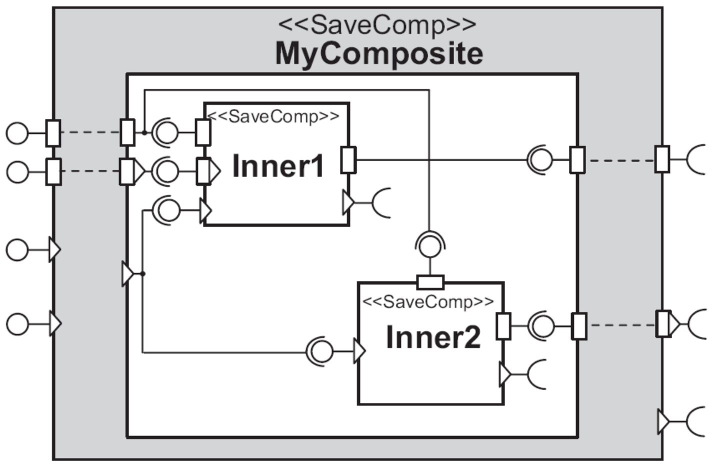
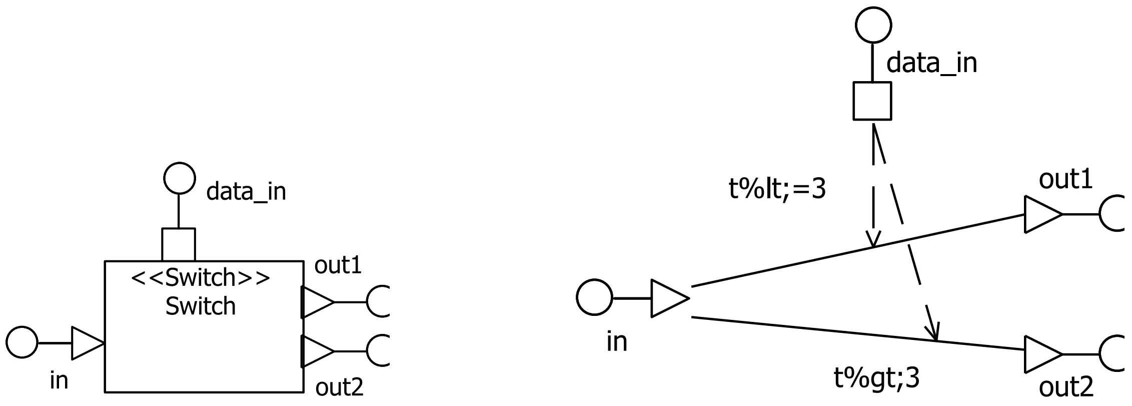


Figure 2.4: Graphical representation of the composite component (from: [Åkerholm et al., 2007b]

example to implement modes and mode switches. A switch consists of a set of internal connection patterns, which partially map the switch's input to its output ports. A logical expression over incoming data guards each of the connection patterns. An example is presented in Figure 2.5.



(a) A switch (b) Inside the switch

Figure 2.5: An example for a switch

As assemblies, switches are not triggered. They instantly react on new incoming data or trigger signals and forward a trigger signal corresponding to the currently active connection pattern. The switches only purpose is the evaluation of the connection pattern guards and perform no other computations.

The SaveCCM switch concept is similar to that in Koala [van Ommering et al., 2000].

2.2.5 Ports

In SaveCCM ports are classified as input or output ports and as trigger ports or typed data ports. All ports have a name. Data ports must also be assigned a type and can have an initial value. The components input ports, the output ports of the system, and the input ports of switches which are used in a connection pattern guard are one-place buffers with overwrite semantics. The data on all further input as well as output ports is not buffered, but passed on immediately.

External ports are ports which are mapped to some external entity, as for example I/O-ports, interrupts, or real-time database pointers. A *combined* port is data as well as trigger port at the same time.

For the connection between ports, their characteristics must be regarded. Trigger output ports may only be connected to trigger input ports. Data output ports have to be consistent with connected data input ports concerning the type. Combined ports can be connected to all three kinds of ports, provided that the data type is compatible.

2.2.6 Connections

SaveCCM distinguishes between immediate and complex connections.

Immediate connections typically exist between components that are located on the same node, which implies loss-less, automatic migration of data or trigger signals from one port to another.

Complex connections represent connections in distributed systems between components on different nodes. They incorporate data and control transfer over channels with possible delay and information loss.

Chapter 3

Design & Implementation

3.1 Mapping SaveCCM Components to Tasks

The objective of this thesis is to design and implement an approach to transform SaveCCM component-based software designs into a runnable format. This implies particularly the allocation of components to runtime tasks. The two most elementary approaches are evidently to either create one task for each component, i.e., a one-to-one allocation, or to map all components to the same task.

A one-to-one allocation can be disadvantageous regarding memory consumption and can also cause overhead due to an increased number of context switches. The allocation of all components to one task on the other hand consumes less memory, but it can lead to a higher processor load in case the components have different periodicities [Fredriksson, 2008]. Since the task's period has to be adjusted to the shortest component period of the co-allocated components, components with longer periods are executed with a higher frequency than required which adds up to a higher load on the processor. The way how components are mapped to tasks has a significant impact on the compliance of extra-functional requirements and should therefore be paid attention to. An elaborated allocation strategy provides the means to find a satisfactory components-to-tasks mapping.

3.1.1 Issues

The allocation strategy affects several extra-functional properties of the system. The number of tasks has impact on memory consumption and the amount of necessary context switches. The choice which components are combined in the same tasks can cause a high processor load.

The requirements can also include isolation sets for components, which denote that certain components may not be allocated to the same task. Furthermore, it can be required that a component is executed on a specified physical node in a distributed system, for example because of sensor data which is available at that particular node and needed for the component's computation.

Most importantly, non-functional requirements within the embedded real-time domain naturally comprehend timing requirements. It is indispensable that the temporal constraints are fulfilled, therefore it is necessary to verify that a components-to-tasks allocation meets the assigned deadlines, i.e., a schedulability analysis needs to be performed. Predictability is vitally important within real-time computing to provide safety assurance.

3.1.2 Strategy

The allocations can be optimized with regards to a series of extra-functional properties, as for example memory consumption or CPU overhead. Finding a by some means optimal allocation is however a highly complex problem. The assumption that the components can be mapped to tasks in an arbitrary way results in an exponential growing search space of possible allocations. Common ways to tackle this problem are probabilistic and heuristic methods, such as simulated annealing or genetic algorithms, as applied in [Fredriksson, 2008]. However, deterministic approaches are preferential in areas where safety, and therefore predictability, are vital issues.

As part of this thesis, an allocation strategy has been developed which consists of a set of deterministic rules. The rules are based on the control flow in the component-based designs and take the triggering mechanism and the precedence and interconnection relations between the components into account.

For the control flow in SaveCCM designs, two trigger types, periodic and event, and four different connection types can be identified. With regard to the further development within the PROGRESS project, a fifth connection type will be added. All five types have been incorporated into the set of allocation rules. The remainder of this section will describe the different connection types and the corresponding allocation rules that have been elaborated.

The Connection Types

The most elementary connection type is the *simple connection*, as shown in Figure 3.1. It connects one component with exactly one succeeding component.

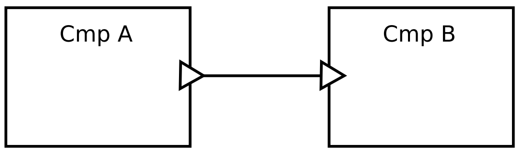


Figure 3.1: A simple connection

A control fork passes a trigger from one component to at least two succeeding components. An example for a *forked connection* with two successor components can be seen in Figure 3.2.

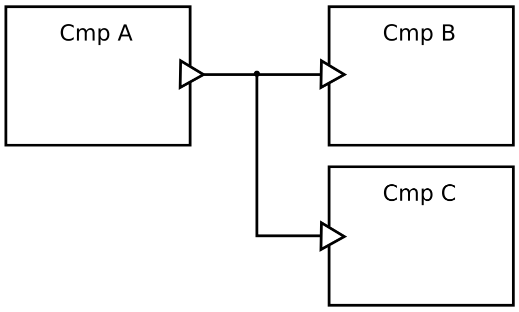


Figure 3.2: A forked connection

A control join or *and-connection* is not explicitly modeled in SaveCCM, but implicitly results from the semantics of the trigger mechanism. As described in Section 2.2.1, a component is trigged when *all* of its trigger input ports are activated. This implies an and-condition for the triggering of a component with more than one trigger input port. Thus, an and-connection connects at least two components with one succeeding component. An example is shown in Figure 3.3. A control join connection will be explicitly introduced with the further development within the PROGRESS project.

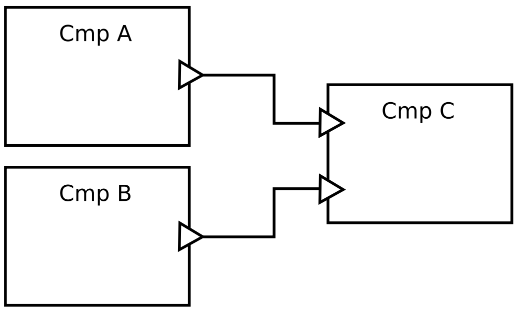


Figure 3.3: An and-connection

The PROGRESS project will also feature a connection type which is not supported by SaveCCM. Analog to the and-connection, an or-connection is introduced. It connects minimum two components with one successor component, which will be triggered as soon as at least one preceding component passes a trigger.

The last connection type results from switches, which are described in Section 2.2.4. In PROGRESS, this concept is referred to as *selection*. A *selection connection* connects one component with an arbitrary number of successor components under a set of conditions. It can be used to establish the actual connection during runtime, based on the current state. An example for a selection connection, implemented by a switch, can be seen in Figure 3.4.

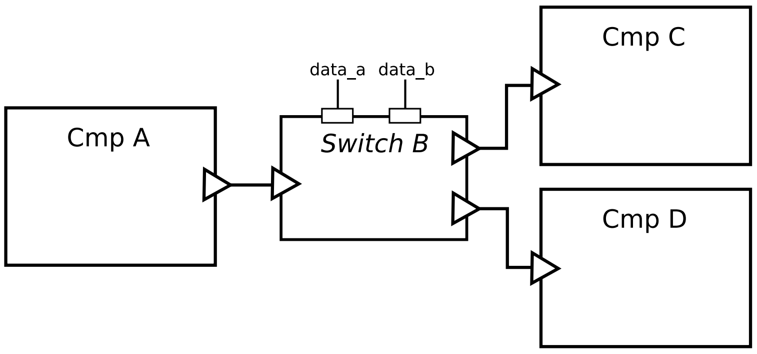


Figure 3.4: A selection connection, instantiated by a switch

The Allocation Rules

For each connection type, an allocation rule has been established. The rules are based on the arrival of triggers and therefore on the activation sequence of components. Chains of connected components can be allocated by sequential application of the rules.

The most elementary case is a single component. It will be mapped to one task, in either case of trigger type. The activation of the task will occur according to the incoming trigger of the component. Every component that is connected to a trigger generator, which can be a clock or an external trigger port, marks the beginning of a control flow thread and each of these threads will be mapped to one task.

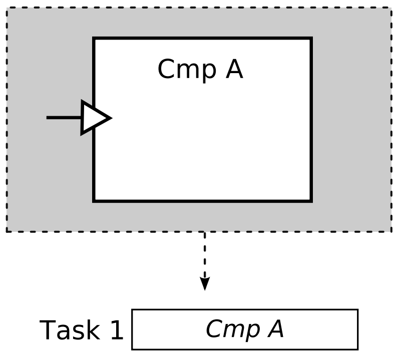


Figure 3.5: Allocating a single component

Two components that are connected with a simple connection will be mapped to the same task, as in Figure 3.6. The execution of component B will always succeed the execution of component A. The trigger of component B is the same as for component A, whether it is event or time triggered.

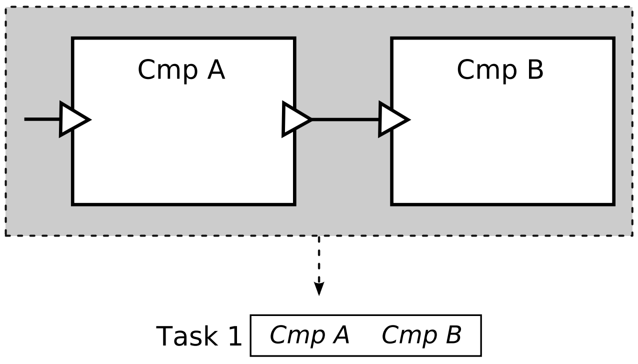


Figure 3.6: Allocating two components with a simple connection

Components that are linked with a forked connection will also be mapped to the same task. An example is shown in Figure 3.7. The components B and C will always succeed the execution of component A and also hold the same trigger. The succeeding components will be placed after the preceding component in a deterministic order. This concept is also valid for forked connections with more than two succeeding components.

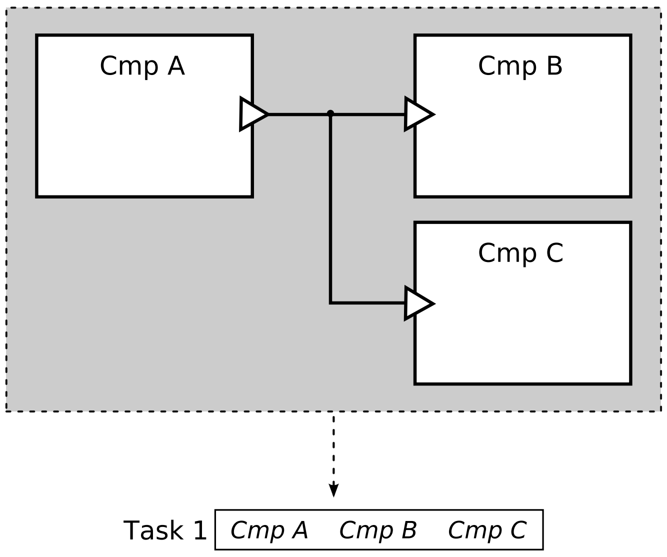


Figure 3.7: Allocating three components with a forked connection

For a component which joins incoming triggers from several preceding components, the allocation varies depending on the combination of trigger types. If all triggers are periodic, i.e., time triggers, the succeeding component will be assigned to the same task as the predecessor with the longest period. This is reasonable since all components with a shorter period will already have been executed and generated a trigger. Therefore it will always be the component with the longest trigger period that delivers the trigger which will fulfill the and-condition and activate the component. This implies that the succeeding component inherits this longest period.

Figure 3.8 demonstrates an example with periodic triggering and two preceding components.

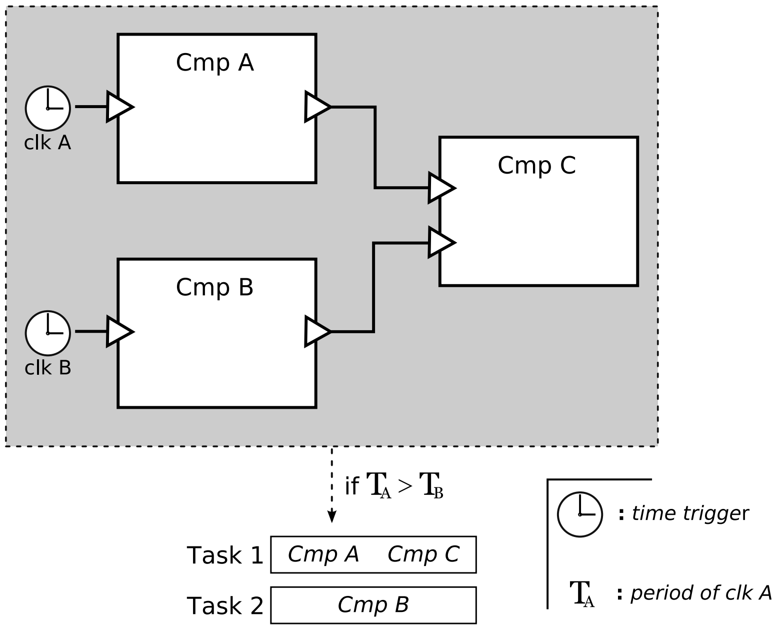


Figure 3.8: Allocating three components with an and-connection and periodic triggers

In the case of only event triggers, as in Figure 3.9, it is not predictable which preceding component will deliver the activating trigger. On this account, the succeeding component will be added to the tasks of all preceding components, together with a guard to check if all other incoming triggers have already been activated. Thus, the component is activated only when all triggers have been set.

The last possible trigger combination, a combination of both time and event triggers, corresponds to the case of only event triggers and will be handled the same way. This results in non-predictable activation times for the preceding components, as in the case of event triggering.

The execution of a component which is attached to an or-connection will always follow each of the preceding components' execution. Hence, the successor component will be mapped to every predecessors' task (see Figure 3.10). In case the incoming triggers include periodic triggers, the shortest period will determine the minimal inter arrival time of an activating trigger for the successor component.

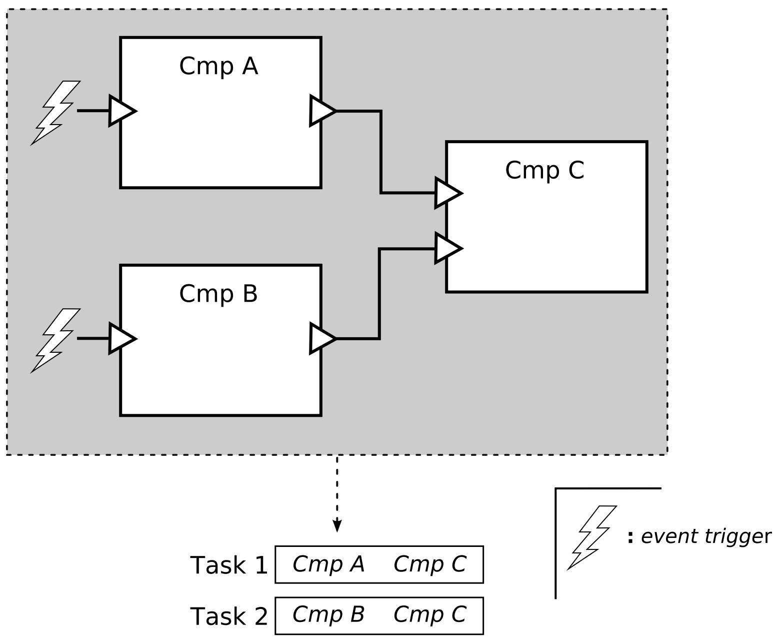


Figure 3.9: Allocating three components with an and-connection and event triggers

The last remaining connection type is the selection connection. A selection connection can be seen as a switchable simple connection. Only one of the succeeding components will be executed subsequent to the preceding component, depending on the condition guarding the connection. Hence, the allocation can be carried out as for a simple connection combined with a conditional expression. All succeeding components will be mapped to the same task as the preceding component. An example can be seen in Figure 3.11.

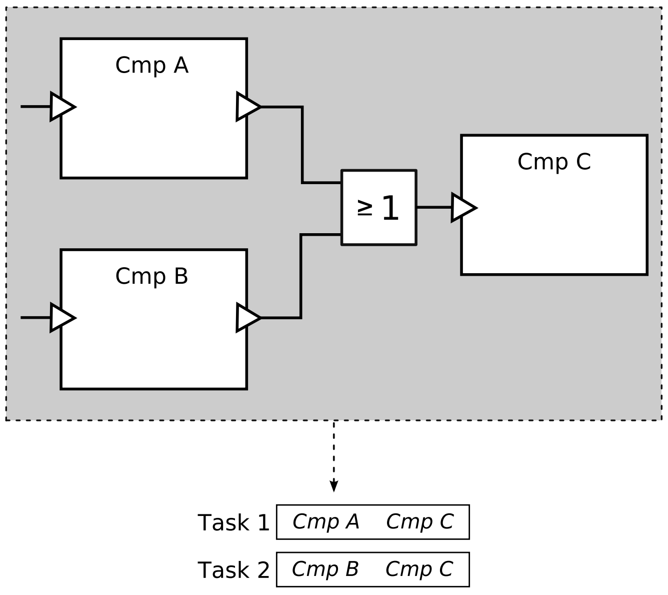


Figure 3.10: Allocating an or-connection

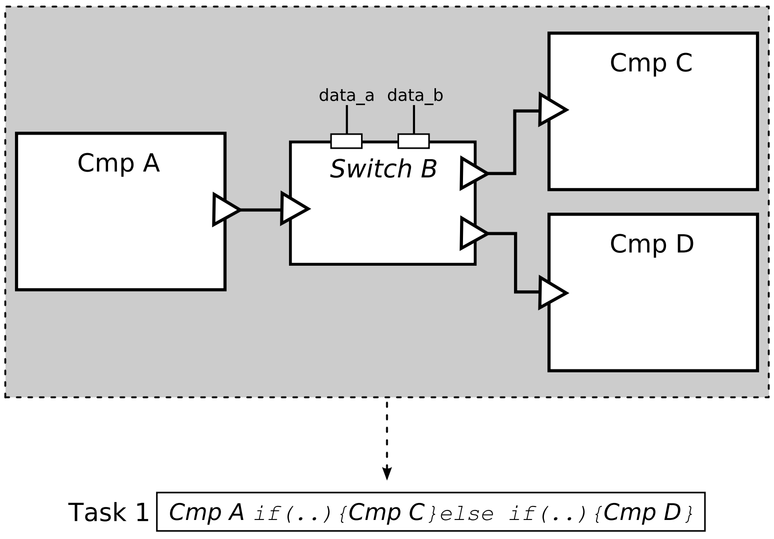


Figure 3.11: Allocating a selection connection

3.2 Implementation

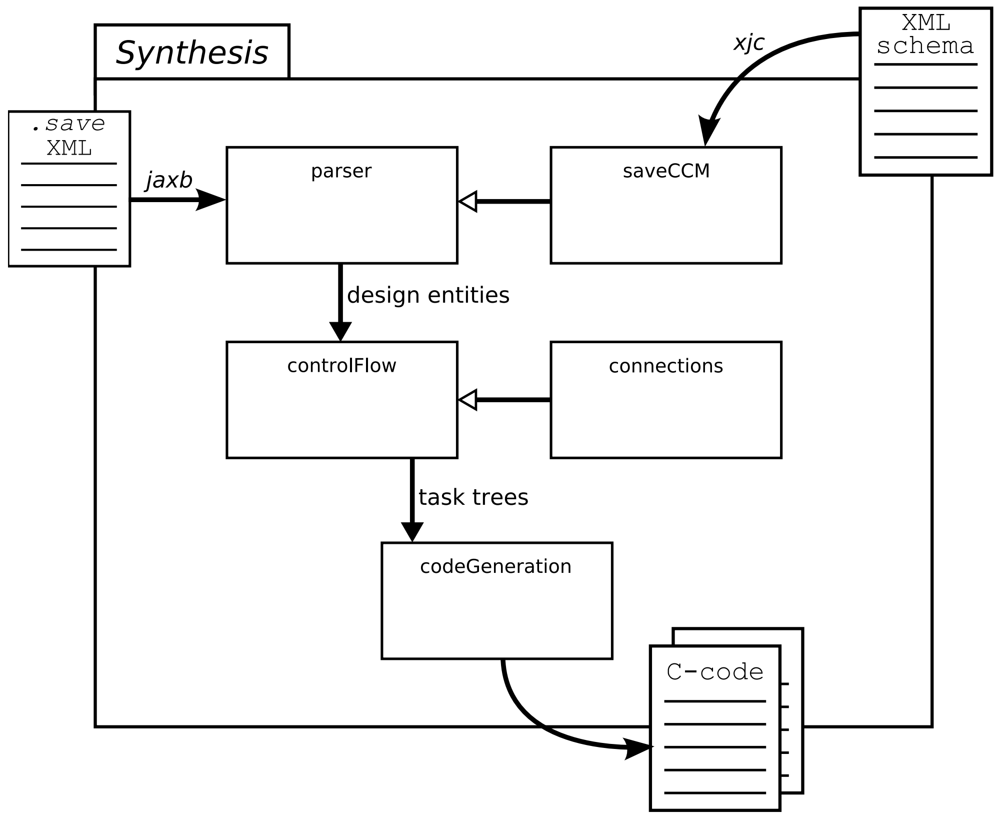


Figure 3.12: Overview over the implementation

The synthesis, i.e., the transformation of SaveCCM component-based designs into an executable format, consists of several consecutive steps. Figure 3.12 gives an overview over the implementation. In the first step, the component-based design is imported and some preprocessing is performed to prepare and facilitate the subsequent steps. Then, the control flow is analyzed and the allocation rules are applied. Finally, the code for the tasks is generated. A standard compilation will conclude the process and translate the generated code into an executable form for a target system. The remainder of this section will describe each of the work packages in detail.

3.2.1 Parsing and Preprocessing

The design which shall be synthesized is delivered as XML file with the extension *.save*. The Java API JAXB[1] has been chosen to retrieve the data from the XML file. JAXB allows the mapping of Java classes to XML and vice versa. Data represented in XML

[1]Java Architecture for XML Binding; for further information see https://jaxb.dev.java.net/

is automatically unmarshalled into Java objects. The included tool `xjc` generates the corresponding Java classes from a given schema file written in XML schema.

A specification of the SaveCCM schema in XML DTD is available in [Åkerholm et al., 2007b]. For the use with JAXB, it has been converted into XML schema. The resulting schema is enclosed in Appendix C. The classes, which have been automatically generated from the schema, can be found in the package `saveCCM` in the source folder.

In addition to the import of the design, certain preparative steps are carried out within this work package. All entities, such as components, switches etc., are sorted into `HashTables` to provide easy access over their ids. Moreover, the connections are converted into an own representation (`Ctrl_Connection`), which enables flexibility. This implementation has been designed for SaveCCM, but also with a view to the extension to ProSave in the PROGRESS project. By wrapping all information which is necessary for the succeeding steps into independent classes, it is possible to adjust the synthesis for different schemas by adapting merely this first step.

Furthermore, assemblies are processed at this point. Since they are solely a mechanism to encapsulate an internal structure and do not feature any behavioral semantics, the contents of an assembly are simply brought to the same level as the rest of the system. This way, no parts of the system are "hidden" for the analysis of the entire control flow.

The corresponding code for the parsing and preprocessing step is avaible in the `parser` package in the `src` folder. The documentation of the package and its functions can be found in `srcDoc` folder.

All information about the design which has been assembled during the parsing and preprocessing is passed to the next step, where the control flow will be analyzed.

3.2.2 Control Flow Analysis

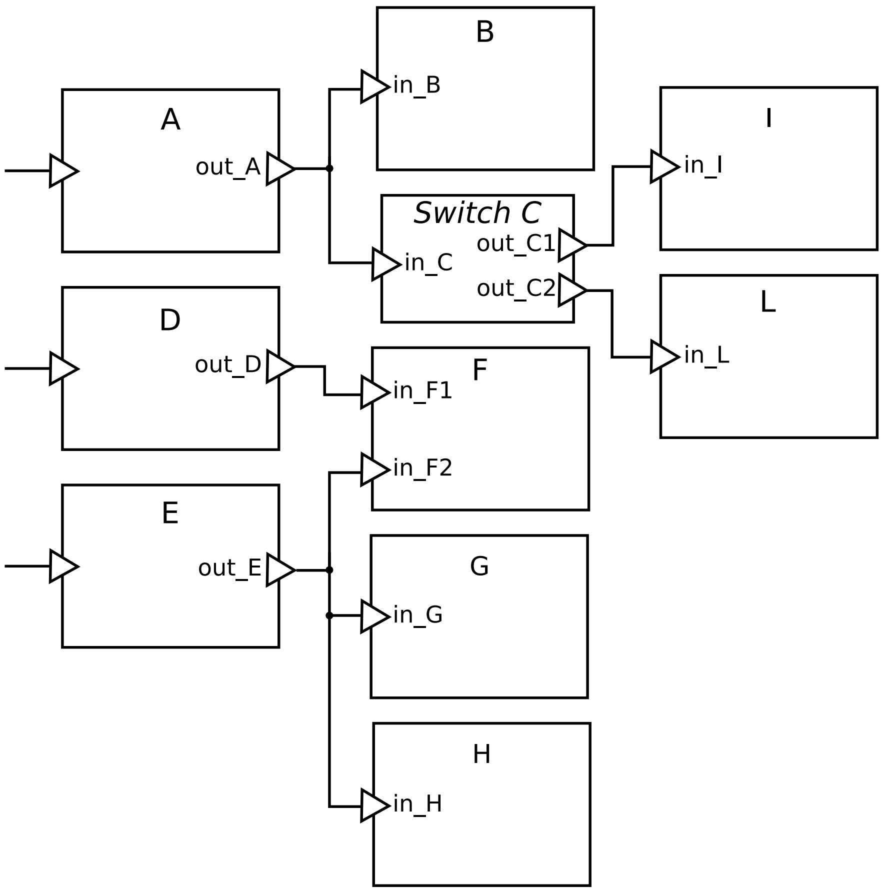
The main part of the synthesis is carried out in this step. First, the control flow is investigated, so that subsequently the mapping of components to tasks can be performed by applying the allocation rules. The control flow consists of all entities that handle triggers, such as trigger generators and connections to and from trigger or combined ports. Since trigger and combined ports bear the same meaning for the control flow, they will be summarized as trigger ports within this section to enhance the readability. The code implementing the tasks decribed in this section can be found in the `controlFlow` package in the `src` folder. The corresponding documentation is available in the `srcDoc` folder.

The analysis of the control flow starts with the identification of the beginnings of control flow threads, in the implementation referred to as `StartingPoints`. A starting point is instantiated by a trigger generator and can either be a clock component, as described in Section 2.2.1, or an external event, i.e., a connection to an external trigger port. Each component connected to a starting point marks the beginning of one control flow thread and accordingly implies the creation of one task.

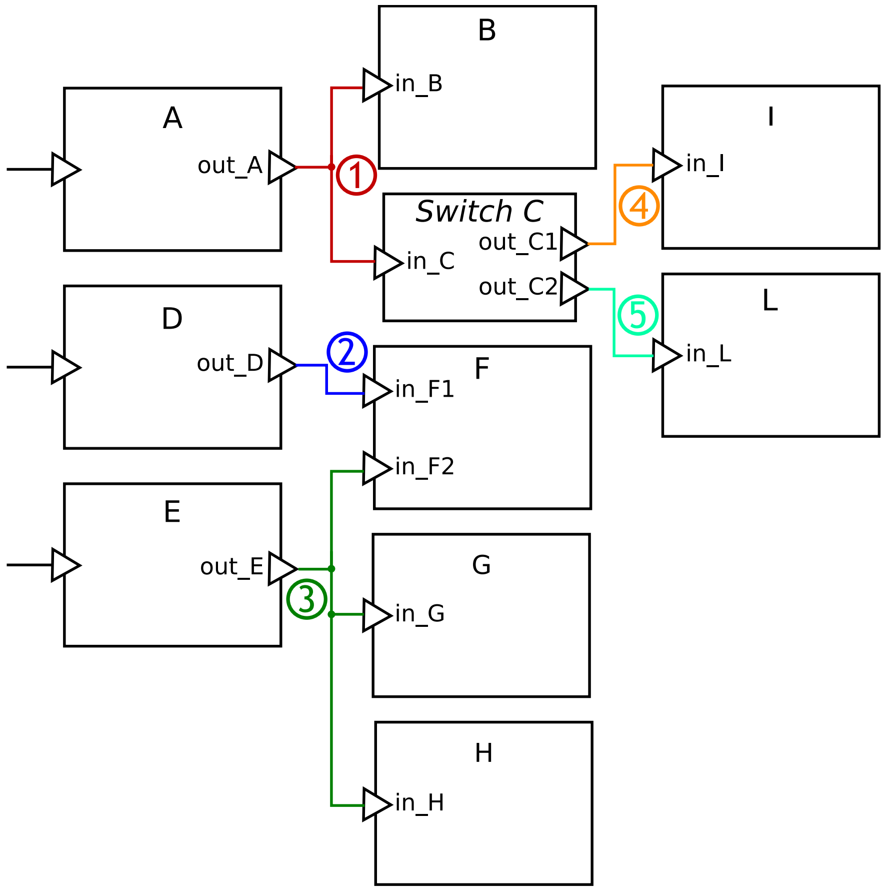
Next, the connection types are identified. In the SaveCCM schema, connections are composed of exactly one origin, represented by the source element's id and the corresponding outport id, and an arbitrary number of destinations, each represented by the target element's id and the corresponding inport id. The classes representing the different connection types are enclosed in the package `connections` in the `src` folder. The package contains an abstract class `Connection` that contains a field of the type `Ctrl_Type`, which is used to specify the connection type, and two abstract methods which return the `From`, i.e., the origin, and `To`, i.e., the destination, respectively. All classes for the connection types are derived from the abstract class. This superclass brings the advantage of a generic data type for the connections, which simplifies the maintenance of connections in data structures, as for example lists or tables.

The identification of connection types needs to be performed in a hierarchical order: from selection connections over and-connections to forked connections and finally simple connections. The reason for this order is that connections which appear to be forked or simple connections, because they match the pattern of one origin and more than one or exactly one destination respectively, can actually be part of a selection connection or and-connection. These connection parts have to be eliminated before forked and simple connections can be properly identified. An example which demonstrates the necessity for the hierarchical processing is shown in Figure 3.13.

Figure 3.13(a) shows an example design with several connections and their interconnection. As pointed out in Figure 3.13(b), there are five original connections. When solely considering the data about the connections, connections 2, 4 and 5 appear to be simple connections and connections 1 and 3 forked connections, respectively. However, from the control flow point of view, the design contains merely four connections, namely one of each connection type. As indicated in Figure 3.13(d), connection 1 is actually not a forked connection, but a simple connection and part of a selection connection. The apparent simple connections 4 and 5 are also part of the selection connection. Moreover, connections 2 and 3 establish a joined connection, which implicates that connection 3 results in forked connection with only two branches. Consequently, all apparent simple connections and branches of forked connection, which are actually part of a selection connection or joined connection, need to be removed before simple connections and forked connections can be correctly identified.



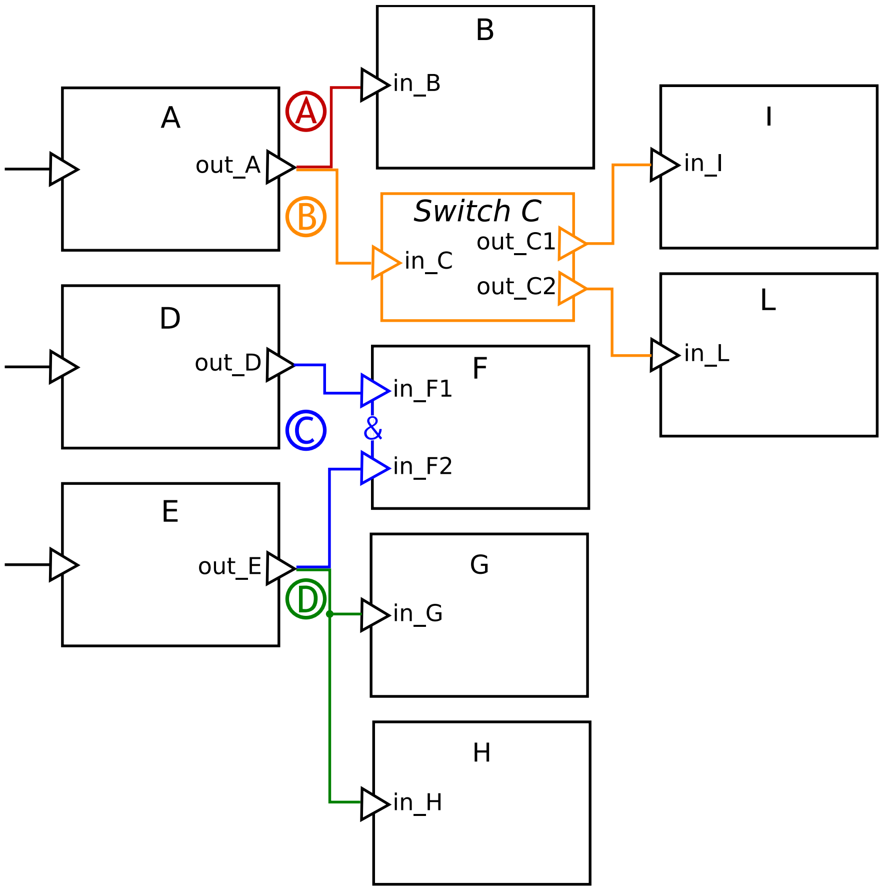
(a) Example connection pattern



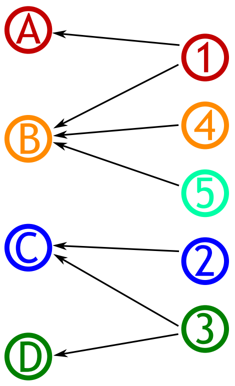
(b) Example with original, untyped connections

#	①	②	③	④	⑤
from to	(A, out_A) (B, in_B) (Switch C, in_C)	(D, out_D) (F, in_F1)	(E, out_E) (F, in_F2) (G, in_G) (H, in_H)	(Switch C, out_C1) (I, in_I)	(Switch C, out_C2) (L, in_L)

(c) Table of connections



(d) Example with determined connection types



(e) Relation between original and determined connections

#	Ⓐ	Ⓑ	Ⓒ	Ⓓ
type	*simple*	*selection*	*joined*	*forked*
from	(A, out_A)	(A, out_A)	(D, out_D) (E, out_E)	(E, out_E)
to	(B, in_B)	(I, in_I) (L, in_L)	(F, in_F1)	(G, in_G) (H, in_H)

(f) Table of determined connection types

Figure 3.13: Hierarchical determination of connection types

To determine selection connections, all switches within the application are considered. According to the schema (see Figure 3.14(a)), a switch contains a series of `SwitchConditions`, each of which consists of an origin and a set of destinations each combined with a conditional guard. From every `SwitchCondition`, a selection connection is created. Since the switch describes its internal connection pattern, the origins and destinations correspond to its inports and outports respectively. For a connection however, the originin is expected to be a components outport just as the destination has to be a component's inport. Hence, the connections to and from the switch need to be found and assigned adequately. Figure 3.14 demonstrates how selection connections are created from a switch and its connections.

Figure 3.14(c) shows an example for a set of components which are connected through a switch. The presented switch connects two trigger inports with three outports and it has two setports which deliver the data that is used for the connection guards. The corresponding XML representation is presented in Figure 3.14(d). The conditional connections inside the switch are visualized in Figure 3.14(c). The figure also includes the tables with the connections to and from the switch, which are needed for the creation of a selection connection. Based on the available data, the corresponding selection connections are created, as illustrated in Figure 3.14(e). Since the switch contains two switch conditions, two selection connections are created.

As mentioned before, and-connections result implicitly from the trigger mechanism of a component. The first step to identify and-connections is to find all components with more than one trigger input port. Then, all connection arriving at the component are gathered from the connections table. Are at least two inports connected to a preceding component, an and-connection will be created. In case ports are detected to be unconnected and not more than one trigger actually arrives at the component, the potential and-connection is discarded and the incoming connection will be classified correctly in one of the two remaining steps. An example for the creation of an and-connection is shown in Figure 3.15.

Within the two previous steps, every connection or part of a connection which belongs to a selection or and-connection have been removed from the connections list. Hence, all remaining connections are either forked or simple connections. A connection which has one origin and more than one destination is classified as forked connection, all connections with one origin and exactly one destination are simple connections.

Once the classification of all connections has been completed, the flow of the triggers through the application is traced, beginning from the components that are connected to one of the previously identified starting points. For every succeeding component, the allocation rule which corresponds to the connection type is applied. The tasks are represented by a tree structure, referred to as `TaskTree`, which suits the need to map possible divergent branches in the control flow. The trees are constructed in a way that the nodes stand for connections and the edges represent the components which link the connections.

This is reasonable, because the connections are the main entities in the control flow and hold the information needed to assemble the tasks.

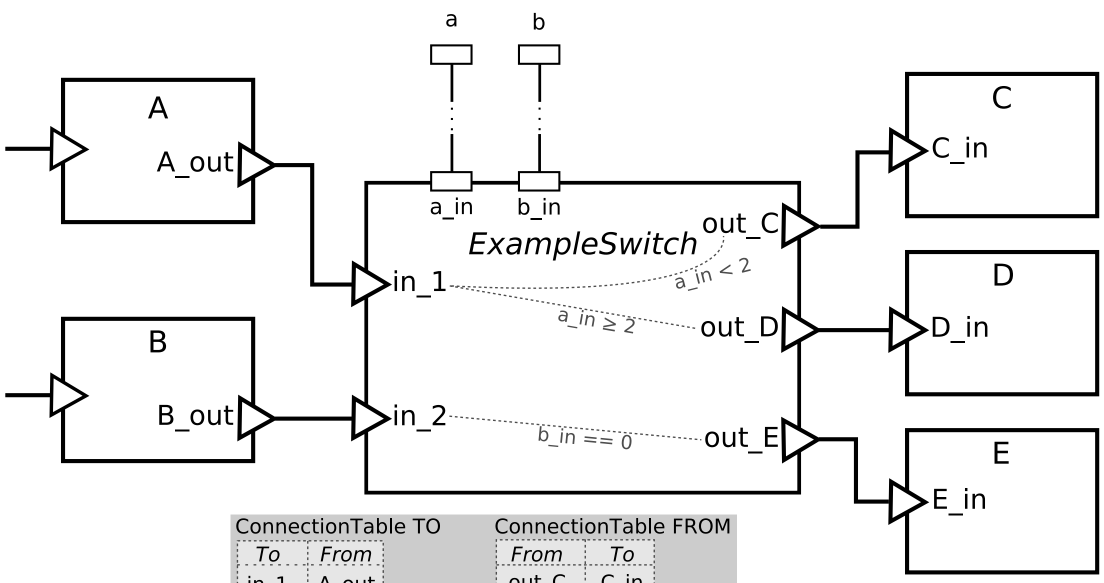
An example for the consecutive application of the allocation rules to construct the task trees is presented in Figure 3.16. Figure 3.16(a) shows an example design with determined connection types. The example contains two starting points, i.e., two clock components in this case, therefore two task trees will be created. The construction of the trees starts with the upper control flow thread. In the first step, as illustrated in Figure 3.16(b), connection 1 is assigned as the root of the first task tree. The root also contains the information about the starting point, such as the period or the external port in case of an event trigger. When a connection has been assigned to the tree, the construction continues with scanning all destination components of the connection for their outgoing connections. In case of connection 1, two outgoing connections, one simple connection (3) and one selection connection (4) are found and assigned as child nodes to the node of connection 1 (see Figure 3.16(c)), according to their allocation rules. In the next step, a joined connection (6) is encountered. For the application of the allocation rule for joined connections, the trigger type of all incoming connections must be known. That implies that all incoming connections must have been assigned to a task tree, because the trigger of a component within a task tree does not necessarily correspond to the trigger of the task. Certain connections, as selection connections and joined connections, can turn a periodic trigger into a sporadic trigger. In the case of connection 6, not all incoming triggers are known yet. Therefore the tree construction proceeds with the second control flow thread. After connection 2 is assigned as the root of tree 2, connection 6 is encountered again. Now all incoming triggers are known and the allocation rule for the joined connection can be applied. Both tasks have periodic triggers, but since tree 1 contains a selection connection before connection 6, the trigger from that tree is actually sporadic. Thus, connection 6 is assigned to both trees, according to the rule for joined connections with mixed triggers. In the last step, simple connection 7 is assigned to both task trees, since it succeeds connection 6.

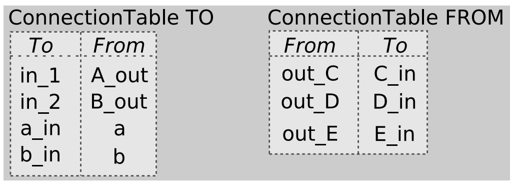
```
<!ELEMENT SWITCHDESC (INPORT*, OUTPORT*, SWITCHCONDITION*)>
<!ATTLIST SWITCHDESC id ID #REQUIRED>
<!ELEMENT SWITCHCONDITION (FROM, (TO, CONDITION)* )>
<!ELEMENT CONDITION EMPTY>
<!ATTLIST CONDITION setport IDREF #REQUIRED value CDATA #REQUIRED>
```

(a) XML schema for switches

```
public class SelectionConnection extends Connection {
  From from;
  List<SelectionCondition> tos;
  String switch_id;
}
public class SelectionCondition {
  List<To> to;
  Condition condition;
}
public class Condition {
    protected String setport;
    protected String value;
}
```

(b) Excerpt from relevant Java classes for a selection connection




ConnectionTable TO

To	From
in_1	A_out
in_2	B_out
a_in	a
b_in	b

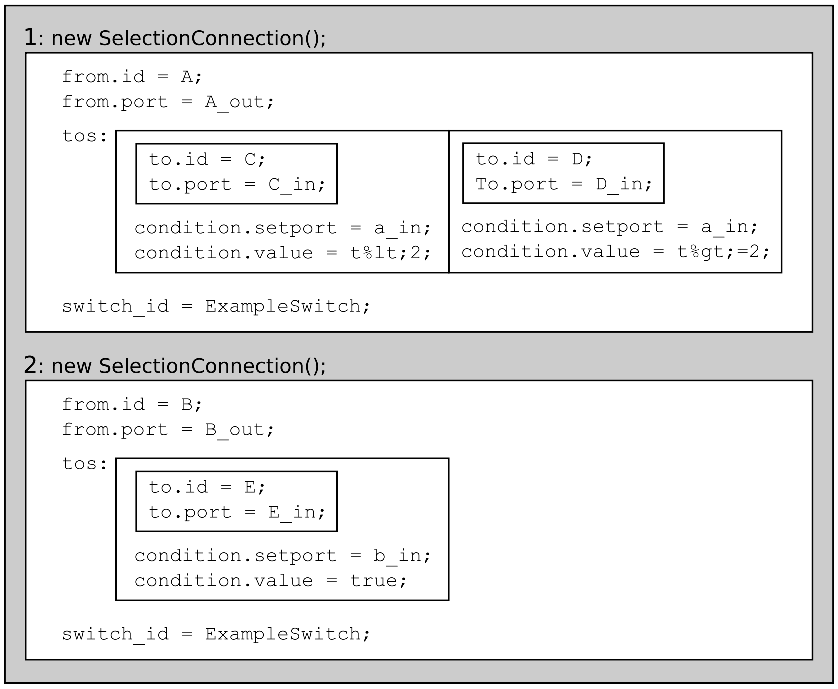
ConnectionTable FROM

From	To
out_C	C_in
out_D	D_in
out_E	E_in

(c) An example switch

```
<SWITCHDESC id="ExampleSwitch">
  <INPORT mode="data" type="int" id="a_in" setport="true"/>
  <INPORT mode="data" type="int" id="b_in" setport="true"/>
  <INPORT mode="trig" id="in_1" setport="false"/>
  <INPORT mode="trig" id="in_2" setport="false"/>
  <OUTPORT mode="trig" id="out_C"/>
  <OUTPORT mode="trig" id="out_D"/>
  <OUTPORT mode="trig" id="out_E"/>

  <SWITCHCONDITION>
    <FROM id="ExampleSwitch" port="in_1"/>

    <TO id="ExampleSwitch" port="out_C"/>
    <CONDITION setport="in_a" value="t%lt;2"/>

    <TO id="ExampleSwitch" port="out_D"/>
    <CONDITION setport="in_a" value="t%gt;=2"/>
  </SWITCHCONDITION>

  <SWITCHCONDITION>
    <FROM id="ExampleSwitch" port="in_2"/>

    <TO id="ExampleSwitch" port="out_E"/>
    <CONDITION setport="in_b" value="true"/>
  </SWITCHCONDITION>
</SWITCHDESC>
```

(d) XML description of the example switch



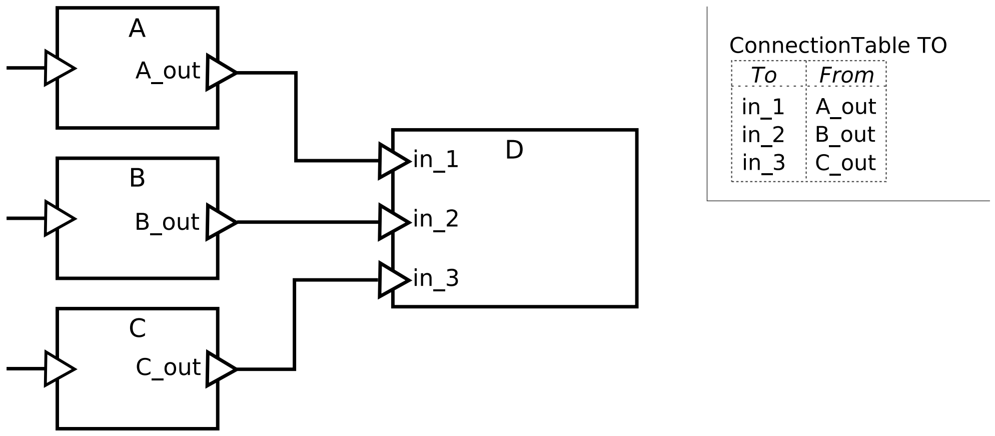
```
1: new SelectionConnection();
  from.id = A;
  from.port = A_out;
  tos:
    to.id = C;
    to.port = C_in;
    condition.setport = a_in;
    condition.value = t%lt;2;

    to.id = D;
    To.port = D_in;
    condition.setport = a_in;
    condition.value = t%gt;=2;
  switch_id = ExampleSwitch;

2: new SelectionConnection();
  from.id = B;
  from.port = B_out;
  tos:
    to.id = E;
    to.port = E_in;
    condition.setport = b_in;
    condition.value = true;
  switch_id = ExampleSwitch;
```

(e) Corresponding Java SelectionConnection objects

Figure 3.14: From a switch to selection connections

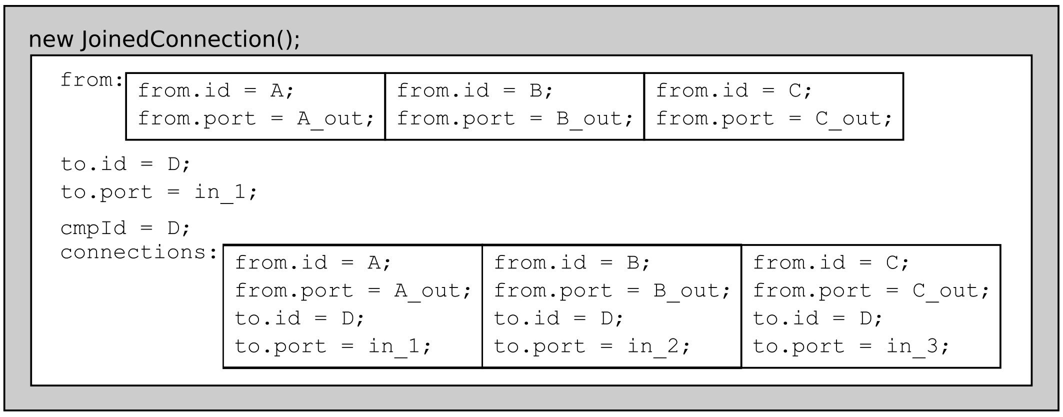
```
public class JoinedConnection extends Connection {
  List<From> from;
  To to;
  String cmpId; //the component where the connection goes to
  List<SimpleConnection> connections; //the initial connection pattern
}
```

(a) Excerpt from the relevant Java class



To	From
in_1	A_out
in_2	B_out
in_3	C_out

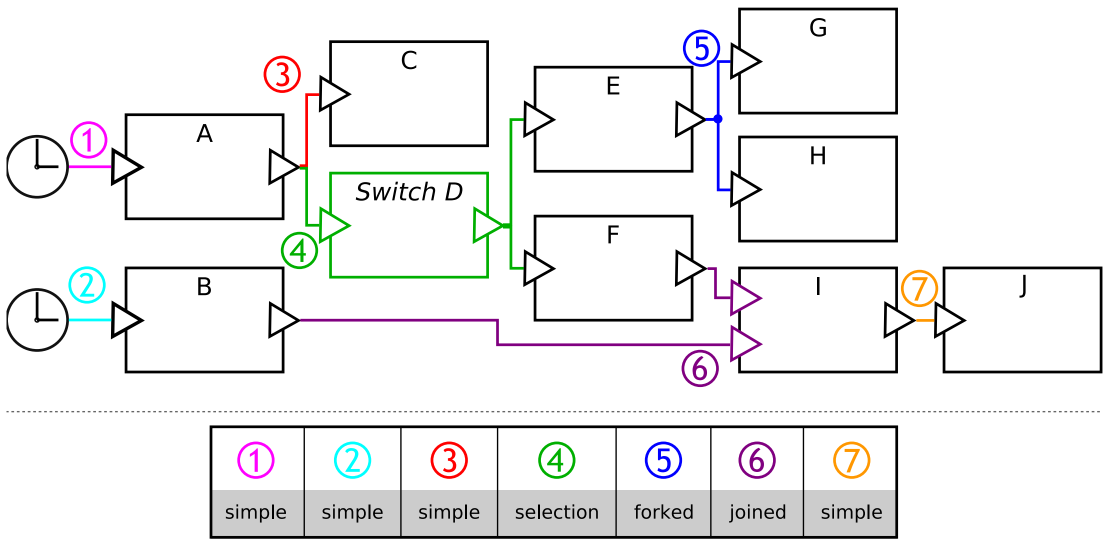
(b) An example for a joined connection



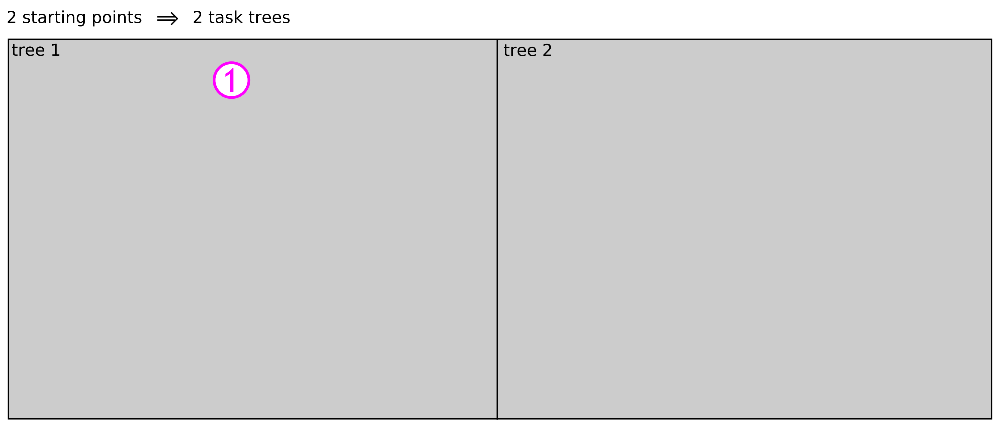
```
new JoinedConnection();
from: from.id = A;          from.id = B;          from.id = C;
      from.port = A_out;    from.port = B_out;    from.port = C_out;
to.id = D;
to.port = in_1;
cmpId = D;
connections: from.id = A;          from.id = B;          from.id = C;
             from.port = A_out;    from.port = B_out;    from.port = C_out;
             to.id = D;            to.id = D;            to.id = D;
             to.port = in_1;       to.port = in_2;       to.port = in_3;
```

(c) Corresponding Java SelectionConnection objects

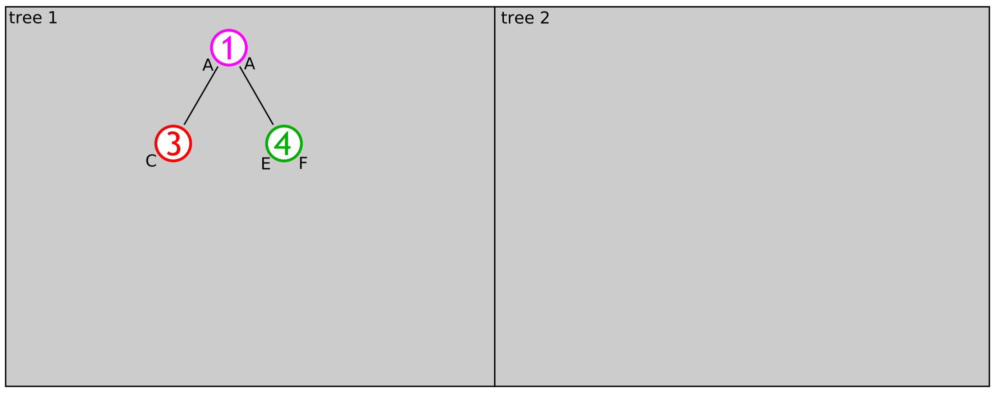
Figure 3.15: Creating an and-connection



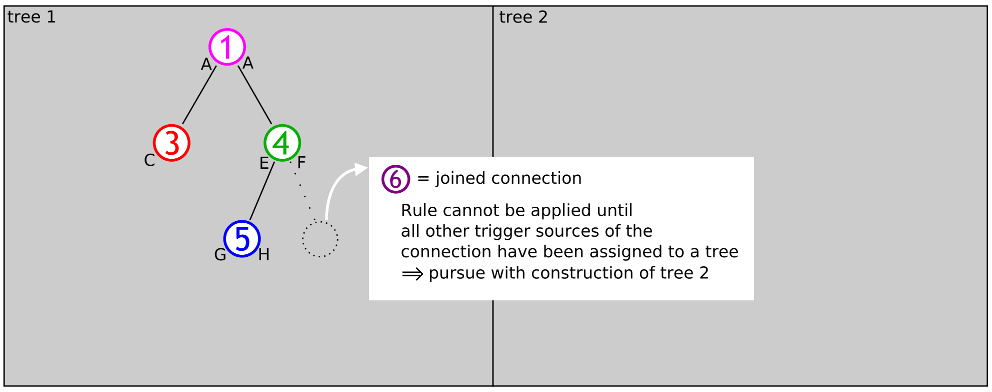
(a) Example with determined connection types



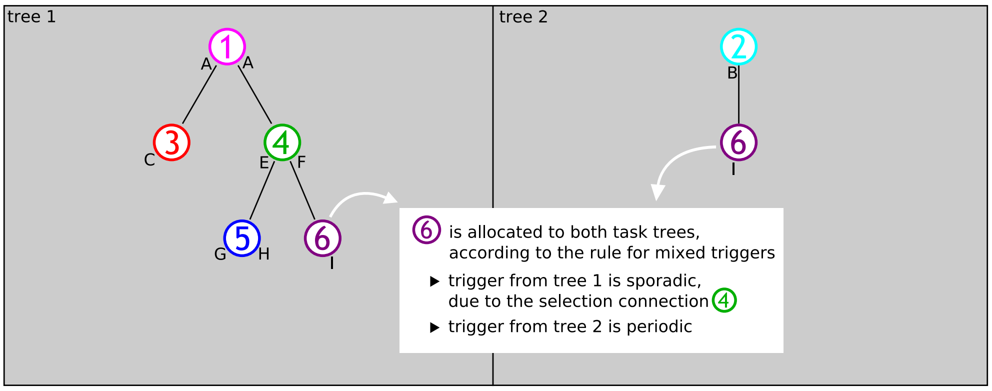
(b) Task tree construction - Step 1



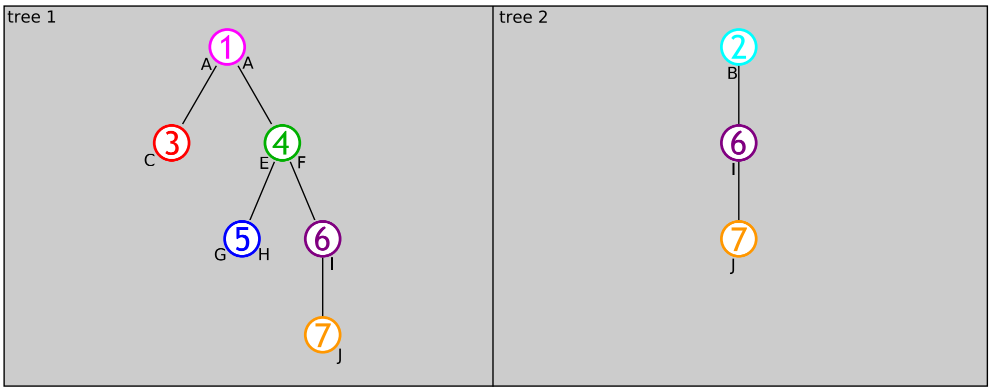
(c) Task tree construction - Step 2


tree 1
tree 2
1
A
A
3
C
4
E
F
5
G
H
6 = joined connection
Rule cannot be applied until
all other trigger sources of the
connection have been assigned to a tree
⟹ pursue with construction of tree 2

(d) Task tree construction - Step 3


tree 1
tree 2
1
A
A
3
C
4
E
F
5
G
H
6
I
2
B
6
I
6 is allocated to both task trees,
according to the rule for mixed triggers
▸ trigger from tree 1 is sporadic,
due to the selection connection 4
▸ trigger from tree 2 is periodic

(e) Task tree construction - Step 4



(f) Task tree construction - Step 5

Figure 3.16: Construction of task trees

When all components have been assigned to a task tree, the allocation is completed and the code for the tasks can be generated.

3.2.3 Task Tree Analysis

With the representation of the control flow in the task trees, the control flow analysis delivers an intermediate result which provides opportunities for further analyses. With further exploration of the task structures, potential for optimization could be revealed and put into effect. An implementation of this step is out of the scope of this thesis, however the potential will be briefly discussed in the following.

Given that deadlines are provided in the application design, the task trees could be utilized to perform a timing analysis. The satisfaction of end-to-end deadlines in the component-based designs could for example be checked, if the execution times of the components have been determined by means of a worst-case execution time analysis prior to the synthesis. In case of a deadline violation, the critical path(s) could be marked in the design to inform the developer about the problem source. Or if possible, attempts could be made to rearrange the components so that the deadlines will be fulfilled.

Even if no deadlines are provided, a timing analysis can be performed, utilizing the components' WCETs, to provide the developer with the timing properties of the designed system.

Furthermore, potential for optimization could result from the internal structure of the tasks. When a control flow thread has been converted into a task tree, the precedence relation between the components in that thread is determined. This fact can be the basis for optimization. A joined connection can for example be simplified, if all incoming triggers are delivered from the same task, which would per definition be the same task that contains the joined connected component. In that case, the code for the task can be simplified, because trigger flags and an if-statement (see Section 3.2.4) for the joined connection will become obsolete. A closer investigation about interdependencies between connection types and their precedence relation could reveal more opportunities for optimization.

3.2.4 Code Generation

The code generation concludes the synthesis. The task trees, which have been created in the preceding step, are transferred into corresponding C code. The source files will be compiled for a target system with a standard compiler in a following step. In the current implementation, the tasks are customized for the real-time operating system (RTOS) *Asterix*[2]. However, a common interface for a series of real-time operating systems, called *SaveOS API*, is created within another project. This will provide abstraction from the operating system and therefore allow for a more general applicability of the task code.

[2]see http://www.mrtc.mdh.se/index.php?choice=projects&id=0026

The implementation of the code generation can be found in the `codeGeneration` package in the `src` folder. The file `CodeGenerator.java` contains the main functionality. The corresponding documentation is available in the `srcDoc` folder.

For each task tree, a header and a source file are created. The names for the files and the task function are composed of the word "task" and a consecutive number, which increases with each task tree. To generate the code corresponding to a task tree, the tree is traversed in level-order, beginning from the root. That means all nodes on one level are visited before the traversal proceeds to the next level. An exception to this is caused by the occurrence of a selection connection. In that case, the execution will not proceed with the component on the same level, since only one of the components after a selection connection is selected for execution. This circumstance leads to one preorder traversal step to first process the entire branch attached to the selected component. After this one step the traversal is resumed in level-order until the next occurrence of a selection connection. The traversal step in preorder also implies the continuation on the level where the selection connection occurred, once the traversal of the branch has been completed. The example in Figure 3.17 clarifies the prodecure.

The components' functionality is provided in the form of entry functions implemented in C. The declarations of the functions and further needed structures, as for example for the combined ports, are generated prior to the synthesis. For all combined ports, a struct with two fields, trigger and value, and the name `inportName_combinedtype` has been created. The path to the folder which contains the source files is passed as an argument to the synthesis. The headerfile that contains the declarations of all entry functions and also all non-standard data types, as for the combined ports, is included at the beginning of each generated source file.

The dataflow is implemented by the use of variables and assigning values to them. To allow for inter-process communication, i.e., the exchange of data between components assigned to different tasks, global variables are created for all outports, which are the sole source for data. As a matter of fact, these variables are only assigned a value in one location, namely the component's entry function. By this means, the data provided by a component in some task will be available in all tasks. Data connections are established by assigning an outport's value to the variable corresponding to the connected inport, as shown in the example in Figure 3.18(c) for both a connection to a simple data port and a combined port.

During the traversal of the tree, for each node the code corresponding to the encountered connection is added to the task source file. The code corresponding to all connection types can be seen in Figures 3.19, 3.20, 3.21 and 3.22. Firstly, the values to the inport variables are assigned. Then, the component's entry function will be called, with the list of inports and outports as parameters.

In the cases of a joined connection or a selection connection, some extra lines of code must be added. For the joined connection, it needs to be checked whether all incoming triggers have been activated. Therefore, each inport which is part of the and-condition has a corresponding global variable. When a component, which is connected to one of these inports, has been executed, the value of the inport variable is set to 1. Prior to the call of the entry function of the and-connected component, an if-statement is added to check whether all trigger input variables have been set to "1". Thus, the component will only be executed in case all triggers have been activated. After the call of the component's function, the trigger variables are reset by assigning the value "0".

For a selection connection, the different branches need to be enclosed in an if-then-else-block, using the conditions guarding the connections. The conditional statements in SaveCCM are represented by a complex type with two elements, `setport` and `value`. The setport holds the data inport of the switch whose value shall be compared. The nature of the comparison is determined by the `value` field. It contains a string of characters, which can express a *lower than [equal]*, *greater than [equal]* or *equal* comparison with a specified constant. In case the `value` string contains the word "`true`", the setport's value is per definition checked for equality with "0". An example for a switch with different conditional statements is shown in Figure 3.22.

To conclude the task creation, a configuration file which describes the properties of all tasks needs to be written for the currently used operating system *Asterix*. An example configuration file is shown in Figure 3.23. For each task, a set of nine properties needs to be specified.

1. HARD_TASK/SOFT_TASK
   specifies whether the task has hard or soft real-time requirements
   possible values: HARD_TASK and SOFT_TASK

2. task activation
   specifies whether the task is activated periodically or sporadically
   possible values: PERIODIC and APERIODIC

3. a name
   assigns a name to the task

4. PERIOD_TIME
   period time in number of time units based on the value of the RESOLUTION field (for periodic tasks)

5. ACTIVATOR
   the signal that activates the task (for sporadic tasks)

6. OFFSET
   the delay of the start within the task's period

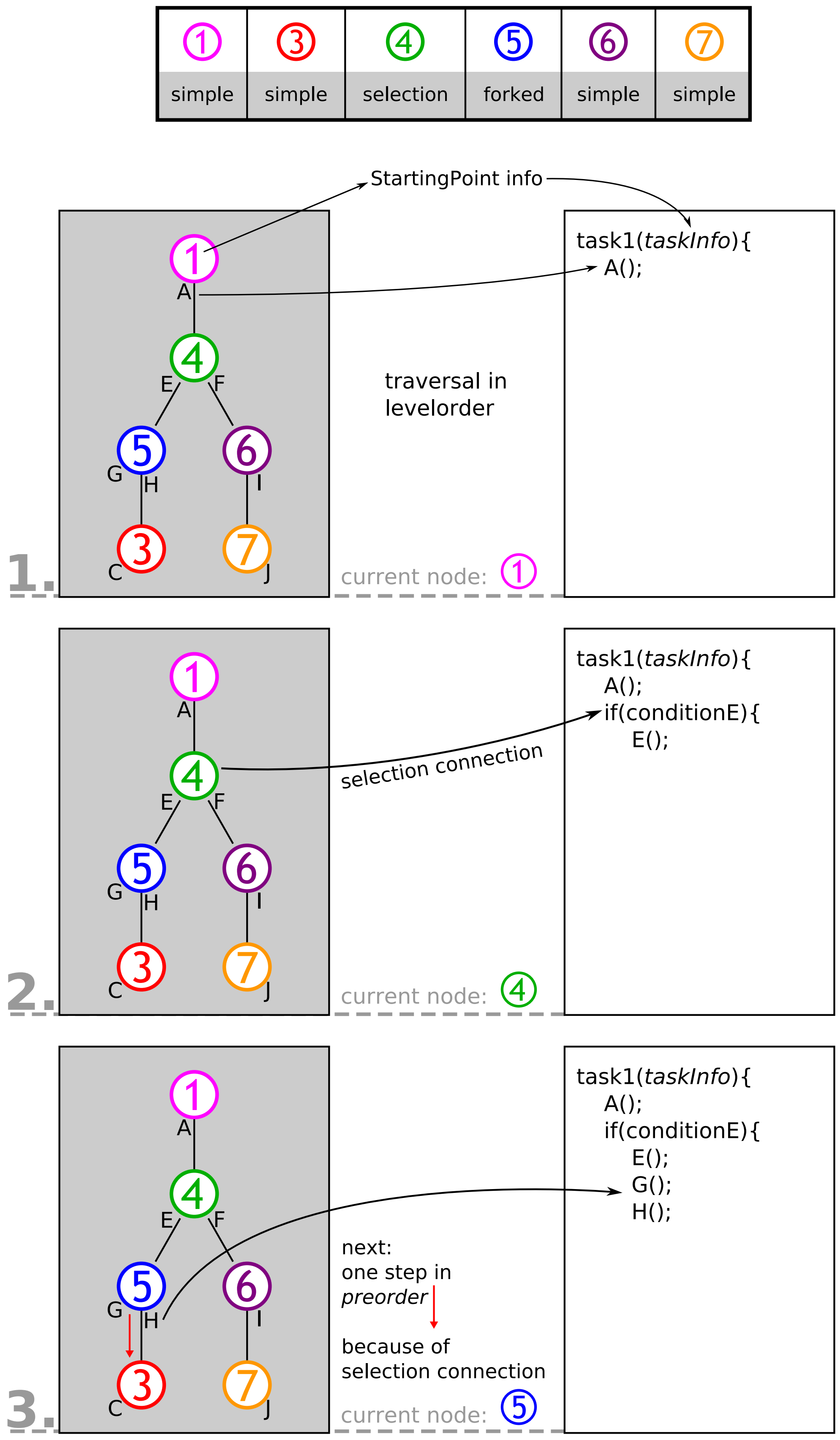
7. DEADLINE
   the latest time for the execution of a task

8. PRIORITY
   the task's priority
   low value = low priority, high value = high priority

9. STACK
   the stack size

10. ROUTINE
    specifies the name of the C-funtion that implements the task's functionality

The information about the task activation, period time and routine name can be directly copied from the task function and task trees. The name of the task's function is utilized as name for the task. All tasks are assumed to have hard real-time requirements. Since neither an offset nor a deadline or priority is currently explicitly specified in the $.save$ file, the offset is set to "0" and the deadline to the period time, which is a safe assumption for the deadline. The priorities are assigned statically according to the Rate Monotonic conventions [Liu and Layland, 1973], i.e., the task with the shortest period gets the highest priority. In case two or more tasks hold the same period time, the order is determined by the task number, which is part of the task name. Also the stack size is not specified by any means and will be set to a default value for all tasks.

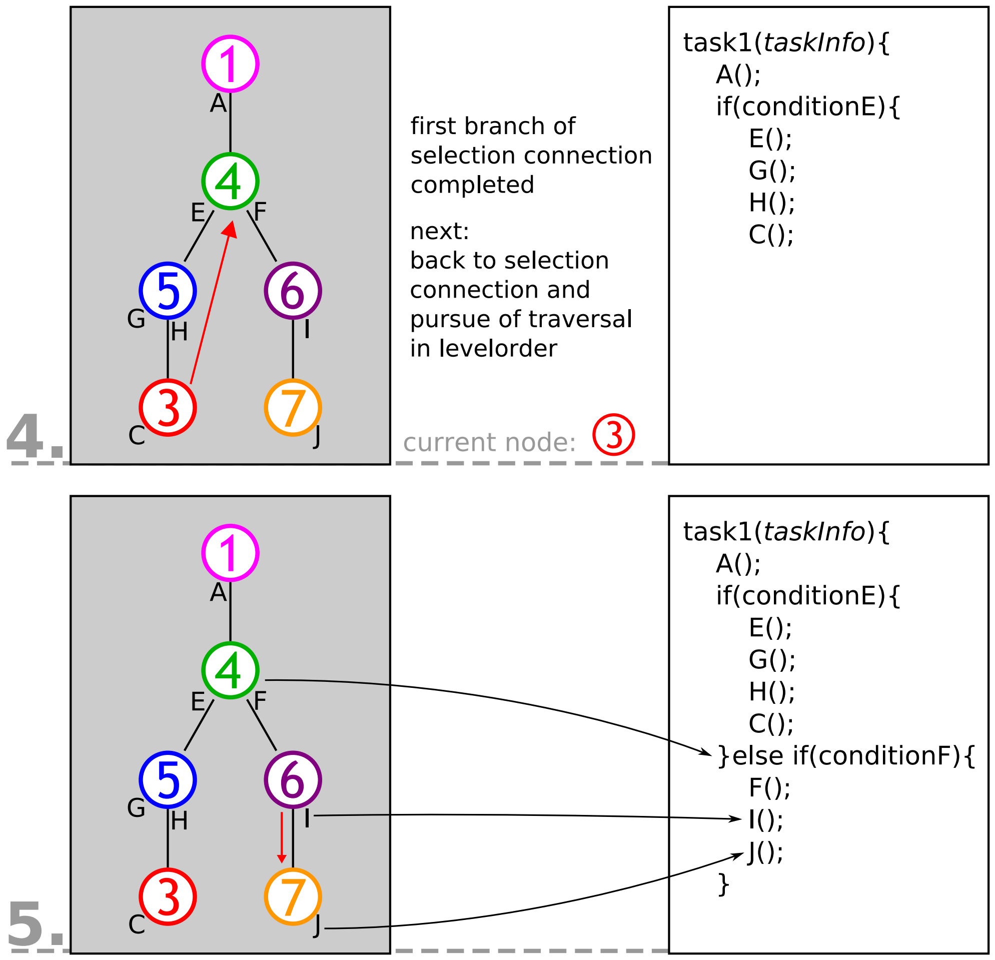


```
task1(taskInfo){
   A();
```

```
task1(taskInfo){
   A();
   if(conditionE){
      E();
```

```
task1(taskInfo){
   A();
   if(conditionE){
      E();
      G();
      H();
```

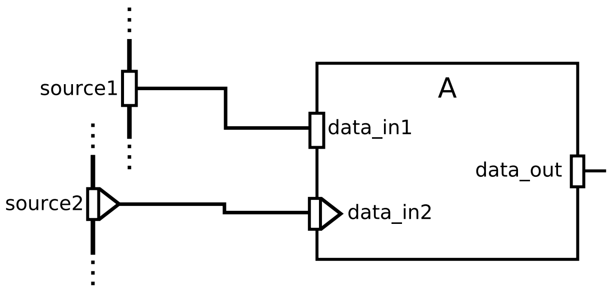
Figure 3.17: Traversal of task trees (1/2)


4.
1
A
4
E
F
5
6
G
H
I
3
7
C
J
first branch of selection connection completed
next: back to selection connection and pursue of traversal in levelorder
current node: 3
5.

```
task1(taskInfo){
   A();
   if(conditionE){
      E();
      G();
      H();
      C();
```

```
task1(taskInfo){
   A();
   if(conditionE){
      E();
      G();
      H();
      C();
   }else if(conditionF){
      F();
      I();
      J();
   }
```

Figure 3.17: Traversal of task trees (2/2)



(a) An example component

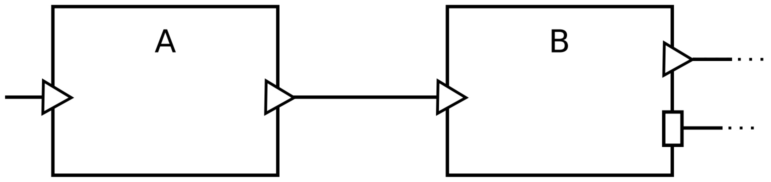
```
int source1;
source2_combinedtype source2;
someDataType data_out;
```

(b) Declaration of global Variables

```
void task(void *ignore){
  int data_in1 = source1;
  data_in2_combinedtype data_in2;
  data_in2.value = source2.value;
  entryfct_A(data_in1, data_in2, &data_out);
}
```

(c) Example code

Figure 3.18: Code generation example

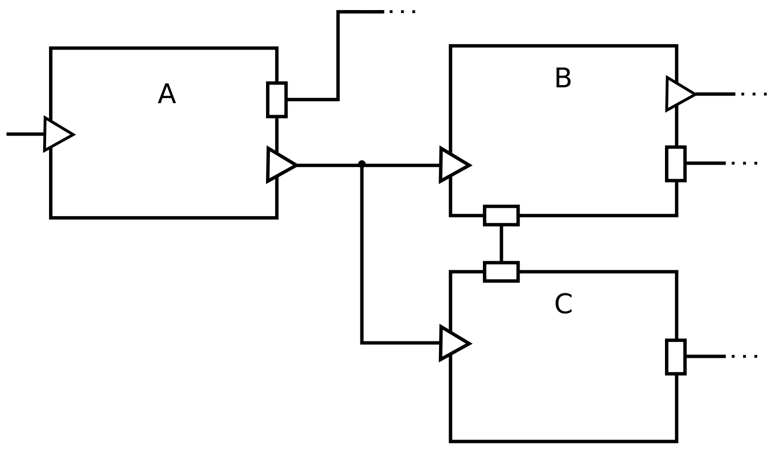


(a) Example for a simple connection

```
void task(void *ignore){
  entryfct_A( /*inports*/, /*list of outports*/ );
  entryfct_B( /*inports*/, /*list of outports*/ );
}
```

(b) Code example for simple connection

Figure 3.19: Code generation example for a simple connection

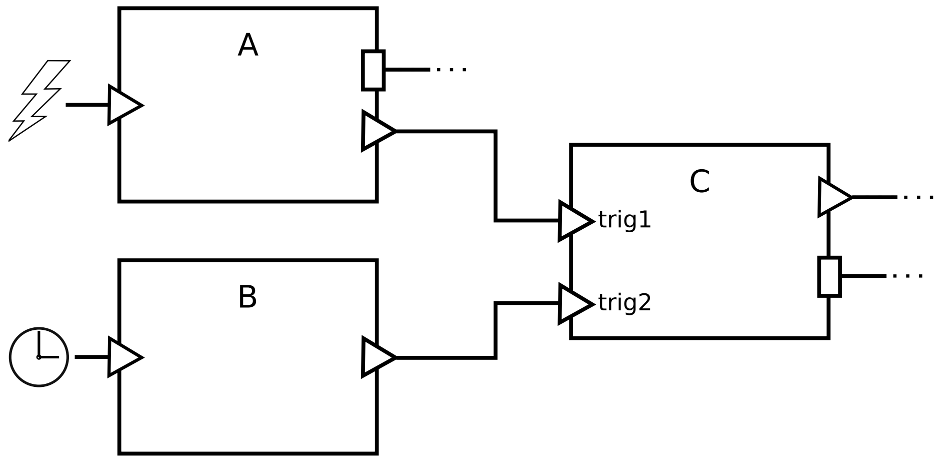


(a) Example for a forked connection

```
void task(void *ignore){
  entryfct_A( /*inports*/, /*list of outports*/ );
  entryfct_B( /*inports*/, /*list of outports*/ );
  entryfct_C( /*inports*/, /*list of outports*/ );
}
```

(b) Code example for forked connection

Figure 3.20: Code generation example for a forked connection

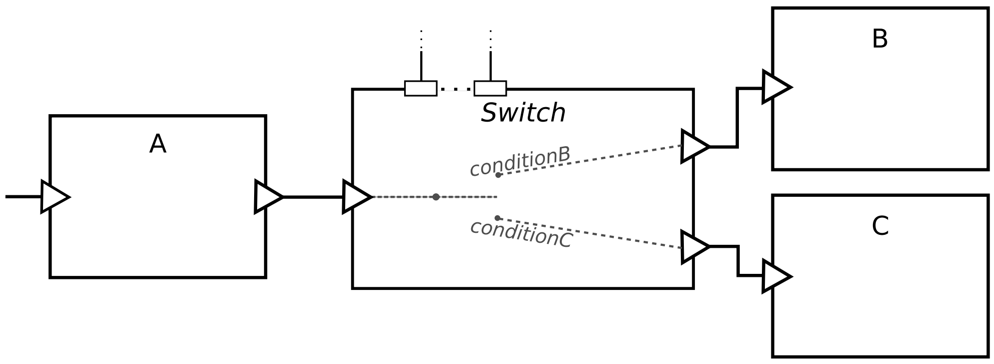


(a) Example for a joined connection

```
//trig1 and trig2 are globally declared

void task0(void *ignore){
  entryfct_A( /*inports*/, /*list of outports*/ );
  trig1 = 1; //activate the trigger input port cmp A is connected to
  if(trig1 == 1 && trig2 == 1) { //if both triggers are active
    entryfct_C( /*inports*/, /*list of outports*/ );
    /*Reset of triggers*/
    trig1 = 0;
    trig2 = 0;
  }
}

void task1(void *ignore){
  entryfct_B( /*inports*/, /*list of outports*/ );
  trig2 = 1; //activate the trigger input port cmp B is connected to
  if(trig1 == 1 && trig2 == 1) { //if both triggers are active
    entryfct_C( /*inports*/, /*list of outports*/ );
    /*Reset of triggers*/
    trig1 = 0;
    trig2 = 0;
  }
}
```

(b) Code example for joined connection

Figure 3.21: Code generation example for a joined connection



(a) Example for a selection connection

```
void task(void *ignore){
  entryfct_A( /*inports*/, /*list of outports*/ );

  if(conditionB){
    entryfct_B( /*inports*/, /*list of outports*/ );
    ... //components following B
  }else if(conditionC){
    entryfct_C( /*inports*/, /*list of outports*/ );
    ... //components following C
  }
}
```

(b) Code example for selection connection

Figure 3.22: Code generation example for a selection connection

```
SYSTEMMODE = NORMAL;
RAM = 65535;

MODE init_mode
{

  RESOLUTION = 10000;

  /* Each task must be configurated here */
  HARD_TASK PERIODIC A{
    PERIOD_TIME = 20;
    OFFSET = 0;
    DEADLINE = 20;
    PRIORITY = 10;
    STACK = 20;
    ROUTINE = a;
  };

  /* There must always be an idle task */
  SOFT_TASK APERIODIC idle{
      ACTIVATOR = 0;
    OFFSET = 0;
    DEADLINE = 0;
    PRIORITY = 0;
    STACK = 100;
    ROUTINE = idletask;
  };

/* Put waitfree-communication here */

/* default signals for irq, do not edit */

/* Put signals here */

/* Put semaphores here */

};
```

Figure 3.23: Example config file

3.3 Integration Into the SAVE IDE

The SAVE IDE[3] is the implementation of the SaveComp component technology. It is realized as a plugin to Eclipse. Apart from the design tool for SaveCCM applications, it also includes a series of analysis tools. More information is available in [Åkerholm et al., 2007a] and [Sentilles et al., 2009]. The synthesis has been integrated into the SAVE IDE and can be initiated directly out of the design tool. Figure 3.24 shows a screenshot of the IDE. The synthesis is started by selecting it in the displayed menu, which appears on a right click. Prior to the synthesis, the system description (the `.save` file) and the C-template have to be generated.

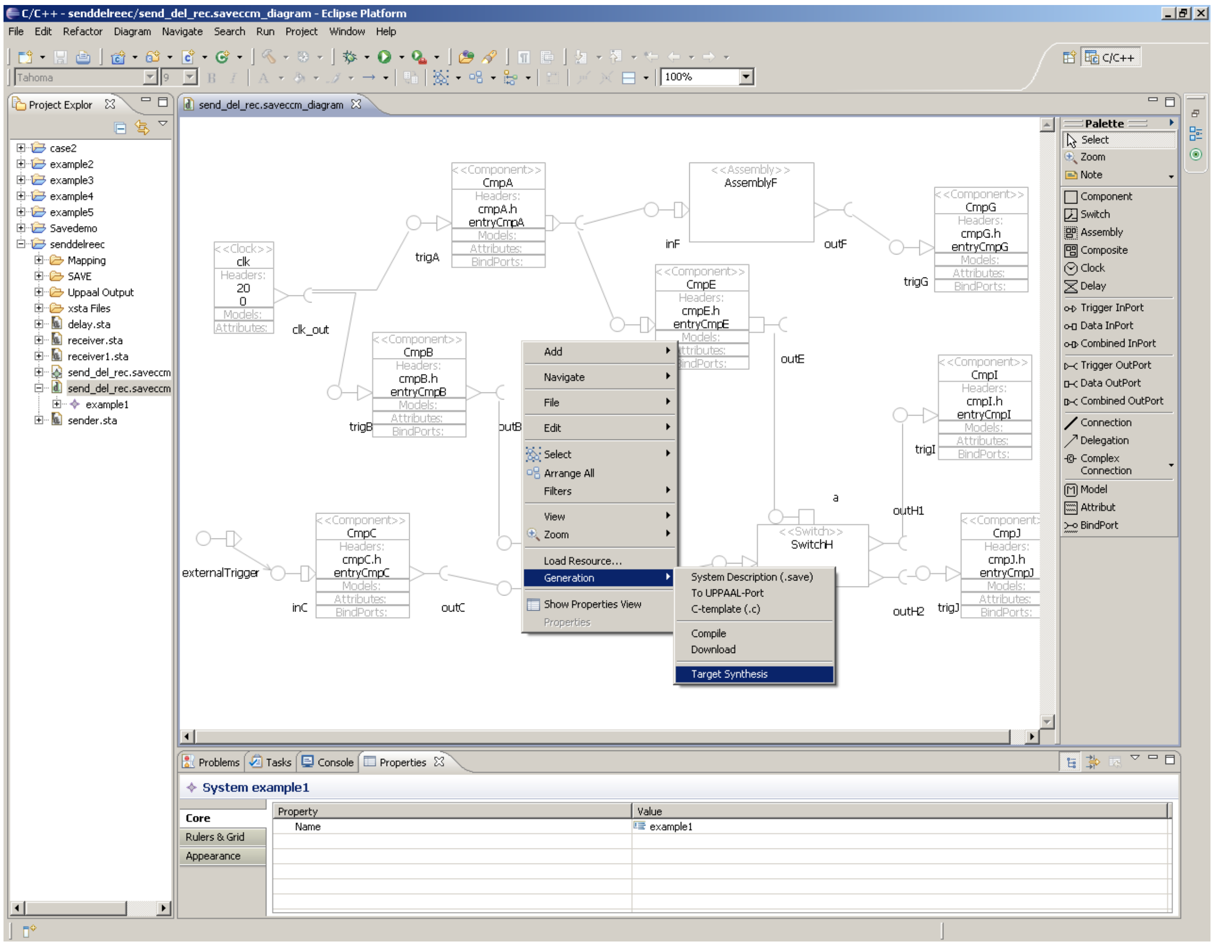


Figure 3.24: Screenshot of the SAVE IDE

[3]Download & installation guide is available at http://sourceforge.net/project/showfiles.php?group_id=206364

Chapter 4

Evaluation & Results

This chapter is concerned with the results which have been obtained from the work within the thesis. The first section describes how the presented implementation has been tested, whereupon the achieved results are presented.

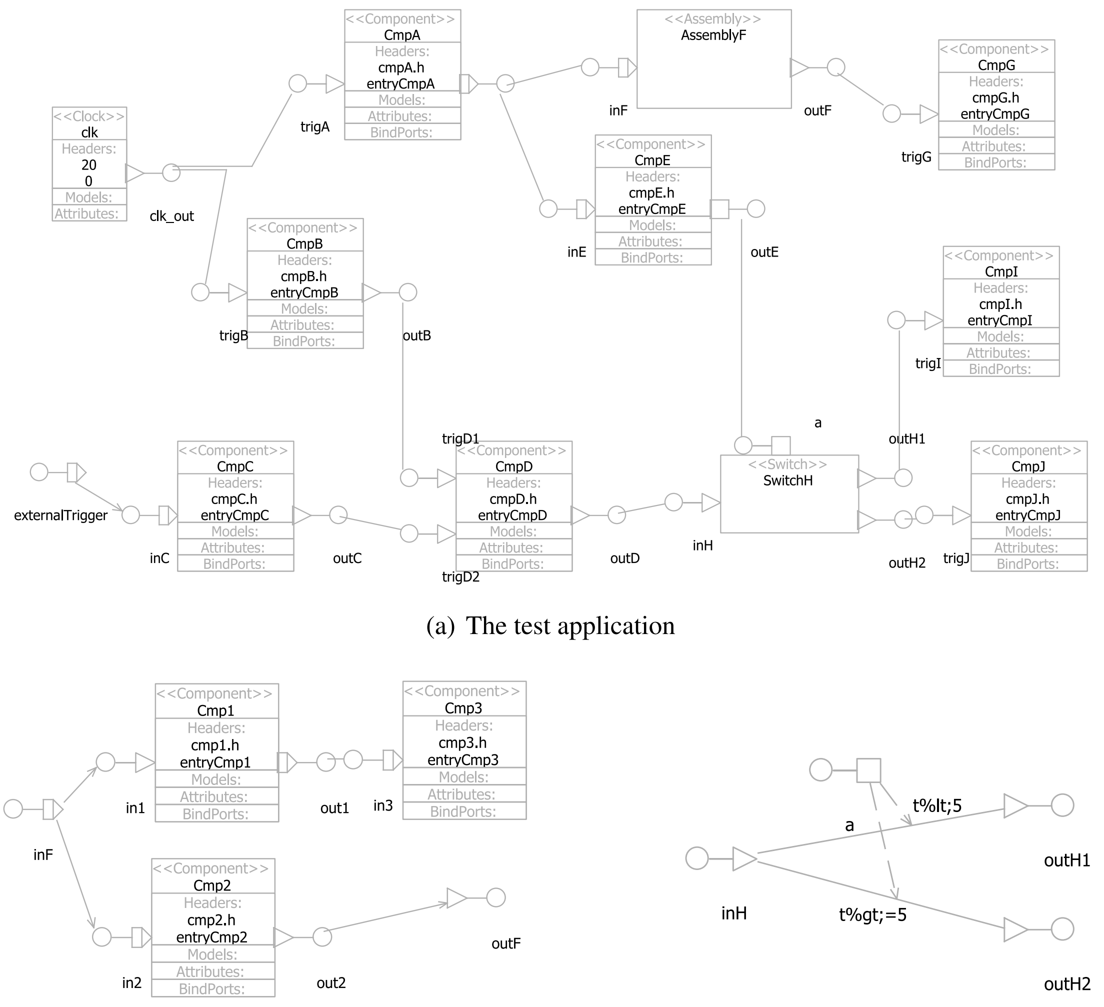
4.1 Testing & Evaluation

This section deals with the testing of the implementation, which has been described in Section 3.2. First, the results produced by each work package of the synthesis will be presented by means of a general test case, which covers all relevant elements of SaveCCM. Thereupon, a number of special test cases will be investigated.

4.1.1 A General Test Case

Figure 4.1 shows a sample SaveCCM application. The corresponding XML representation can be found in Appendix D. The example contains:

- ▷ three starting points, covering both trigger types
- ▷ one assembly with three internal components
- ▷ one switch
- ▷ a total of twelve components
- ▷ at least one connection of each connection type, specifically:
  - ▸ three simple connections
  - ▸ two forked connections
  - ▸ one joined connection
  - ▸ one selection connection



(a) The test application

(b) Inside AssemblyF

(c) Inside SwitchH

Figure 4.1: An example application for testing

The following sections will present the results which are produced by each of the three packages when applying the synthesis to the example.

The `parser` Package

The task of the parser package is to correctly unmarshal the information about the SaveCCM design which shall be deployed from the `.save` file. To visualize the outcome of the parsing step, a list containing all encountered elements has been printed into a file. As shown in Figure 4.2[1], all elements contained in the example have been imported properly. Hence, the `parser` package satisfies its fuctional requirements.

[1]The "#", followed by an integer is added to the elements' names during the generation of the `.save` file to provide for unambiguous names

```
Application name: example1

Components:
  Component CmpA#4
  Component CmpB#7
  Component CmpC#10
  Component CmpD#13
  Component CmpE#17
  Component CmpG#31
  Component CmpI#38
  Component CmpJ#40
  Component clk#2

Switches:
  Switch SwitchH#33

Assemblies:
  Assembly AssemblyF#20Impl

    Components:
      Component Cmp1#23
      Component Cmp3#26
      Component Cmp2#28

    Switches:
      none

    Assemblies:
      none

    Connections
      Connection 1:
        From: AssemblyF#20Impl, port outF#22
        To: CmpG#31Impl, port trigG#32

      Connection 2:
        From: Cmp1#23Impl, port out1#25
        To: Cmp3#26Impl, port in3#27

      Connection 3:
        From: AssemblyF#20Impl, port inF#21
        To: Cmp2#28Impl, port in2#29
        To: Cmp1#23Impl, port in1#24

      Connection 4:
        From: Cmp2#28Impl, port out2#30
        To: AssemblyF#20Impl, port outF#22
```

Figure 4.2: Result of the parser package (1/2)

```
Connections
  Connection 1:
    From: clk#2Impl, port clk_out#3
    To: CmpB#7Impl, port trigB#8
    To: CmpA#4Impl, port trigA#6

  Connection 2:
    From: CmpC#10Impl, port outC#11
    To: CmpD#13Impl, port trigD2#15

  Connection 3:
    From: CmpB#7Impl, port outB#9
    To: CmpD#13Impl, port trigD1#14

  Connection 4:
    From: CmpD#13Impl, port outD#16
    To: SwitchH#33Impl, port inH#34

  Connection 5:
    From: SwitchH#33Impl, port outH1#35
    To: CmpI#38Impl, port trigI#39

  Connection 6:
    From: SwitchH#33Impl, port outH2#36
    To: CmpJ#40Impl, port trigJ#41

  Connection 7:
    From: CmpE#17Impl, port outE#19
    To: SwitchH#33Impl, port a#37

  Connection 8:
    From: example1#0Impl, port externalTrigger#1
    To: CmpC#10Impl, port inC#12

  Connection 9:
    From: CmpA#4Impl, port outA#5
    To: CmpE#17Impl, port inE#18
    To: AssemblyF#20Impl, port inF#21
```

Figure 4.2: Result of the parser package (2/2)

The `controlFlow` Package

The `controlFlow` package has three main objectives:

1. identification of the control flow thread's starting points, i.e., components that are connected to a clock component or an external trigger port

2. determination of all connections' types

3. construction of the task trees, which represent the control flow threads

To show that the `controlFlow` package completes all three steps with the correct outcome, excerpts from the log-file which feature the relevant information are presented in the following. Figure 4.3 contains the part of the log-file which informs about the identified starting points for the control flow threads. All three starting points have been discovered and interpreted correctly.

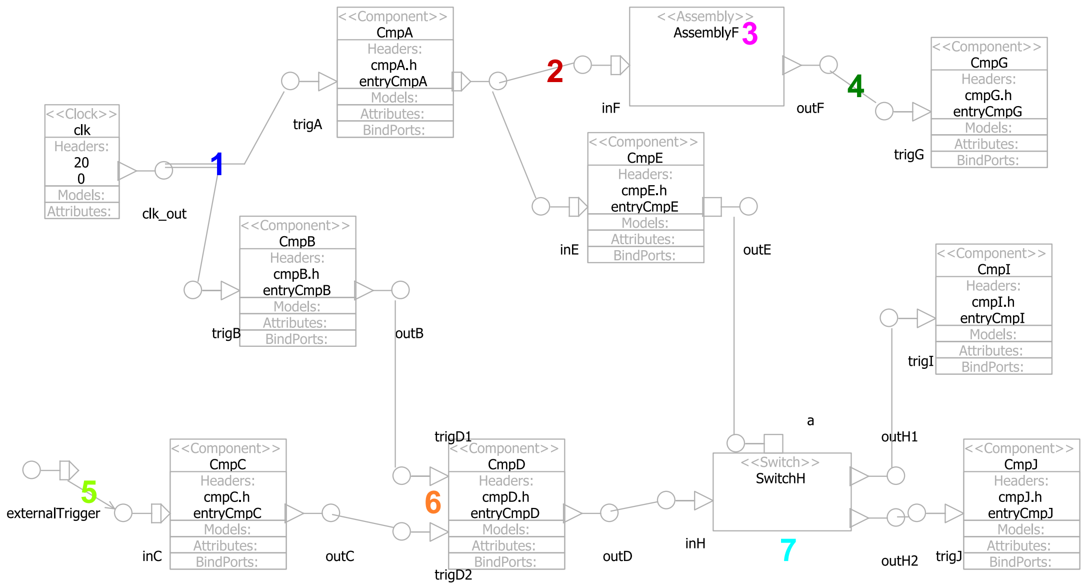
```
Find Starting Points...
    Starting point found!
    Type: CLOCK, component: CmpB#7
    Starting point found!
    Type: CLOCK, component: CmpA#4
    Starting point found!
    Type: EXTERNAL TRIGGER, component: CmpC#10
```

Figure 4.3: The determined starting points

Figure 4.4 presents the determination of connection types that has been performed by the `controlFlow` package. The connections are marked with colored numbers both in the design and the list to depict the correlation. It can be seen that all seven connections have been identified correctly.

Figure 4.5 shows the process of the task tree construction. It includes illustrations that go along with the steps of the construction as captured in the log-file. The numbers representing the tree nodes correspond to those introduced previously in Figure 4.4. During the processing of task tree 0, a joined connection is encountered. Thereupon, the completion of that tree is postponed until all incoming triggers of that connection have been assigned to a tree, as described in Section 3.2.2 (see Figure 3.16). For the present example, this case occurs during the construction of tree 2, whereupon the processing of tree 0 is resumed.

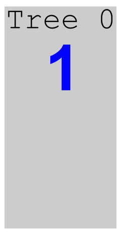
Thus, the `controlFlow` package has successfully completed each of its three tasks.

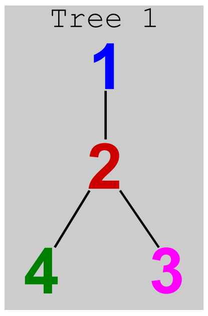


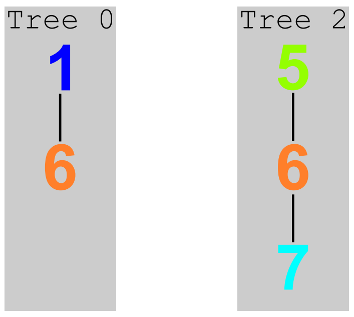
```
Processing Switches..
  7   SelectionConnection found (Switch SwitchH#33):
         From component CmpD#13, port: outD#16
         To component(s):
            CmpJ#40, port: trigJ#41
         under the condition t%gt;=5 ; Setport: outE#19
         To component(s):
           CmpI#38, port: trigI#39
         under the condition t%lt;5 ; Setport: outE#19
Processing JoinedConnections..
  6   JoinedConnection found:
         From: CmpB#7, CmpC#10,
         To: CmpD#13
Processing ForkedConnections..
  1   ForkedConnection found
         From: clk#2
         To: CmpB#7, CmpA#4,
  2   ForkedConnection found
         From: CmpA#4
         To: CmpE#17, Cmp2#28, Cmp1#23,
Processing SimpleConnections..
  5   SimpleConnection found
         From: example1#0
         To: CmpC#10
  3   SimpleConnection found
         From: Cmp1#23
         To: Cmp3#26
  4   SimpleConnection found
         From: Cmp2#28
         To: CmpG#31
```

Figure 4.4: The identified connections

```
Buildung 3 task trees.
Tree 0
  Current component: CmpB#7
  Current outport: outB#9
   ...1 connections found.
   JoinedConnection found!
    ...not complete yet!
Tree 1
  Current component: CmpA#4
  Current outport: outA#5
   ...1 connections found.
   ForkedConnection found!
  Current component: CmpE#17
  Current outport: outE#19
  Current component: Cmp2#28
  Current outport: out2#30
   ...1 connections found.
   SimpleConnection found!
  Current component: Cmp1#23
  Current outport: out1#25
   ...1 connections found.
   SimpleConnection found!
  Current component: CmpG#31
  Current component: Cmp3#26
Tree 2
  Current component: CmpC#10
  Current outport: outC#11
   ...1 connections found.
   JoinedConnection found!
    ...complete -> apply rule
    incoming trigger 1: CLK
    incoming trigger 2: EVENT
    => trigger combi: EVENT/MIXED
     +added to parent node: outB#9
     +added to parent node: outC#11
  Current component: CmpD#13
  Current outport: outD#16
   ...1 connections found.
   SelectionConnection found!
Tree 0
  Current component: CmpD#13
  Current outport: outD#16
   ...1 connections found.
   SelectionConnection found!
Tree 2
  Current component: CmpJ#40
  Current component: CmpI#38
Tree 0
  Current component: CmpJ#40
  Current component: CmpI#38
```







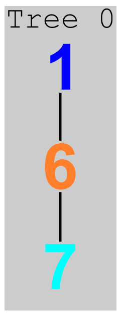


Figure 4.5: Tree construction

The `codeGeneration` Package

The code generation is the last step of the synthesis and produces the aspired final outcome, i.e., C-code which implements the tasks of the designed application. The present example consists of three control flow threads, hence three task source files have been generated. The result source files are presented in Figures 4.7 to 4.9. To facilitate the traceability, Figure 4.6 demonstrates which parts of the application have been assigned to which task. The global variables which are generated for all outports to allow for interprocess communication are listed in Figure 4.10. Figure 4.11 shows the corresponding configuration file for the underlying operating system.

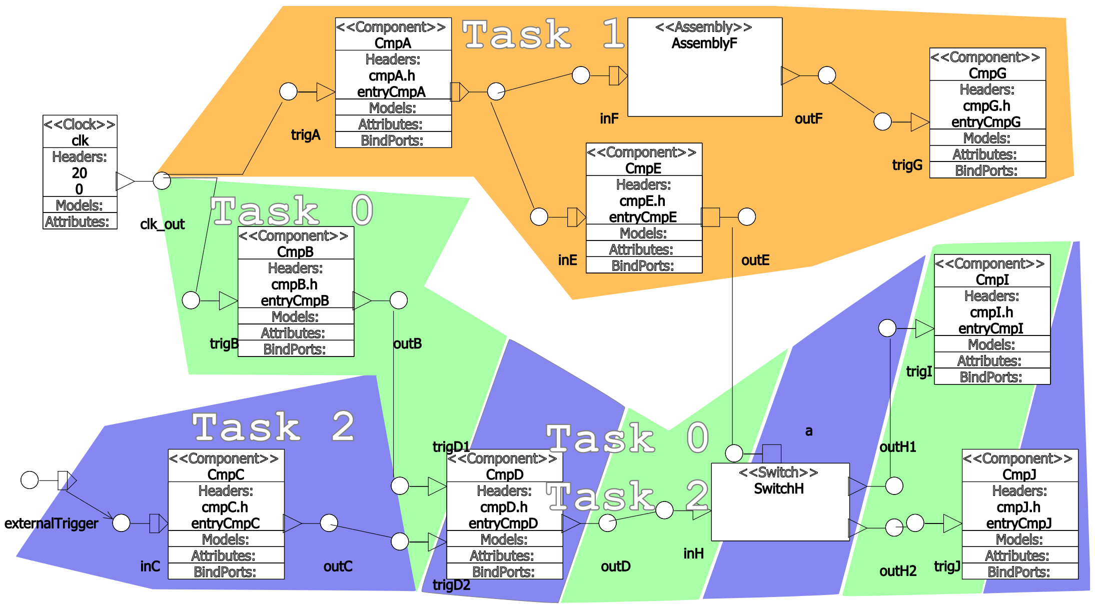


Figure 4.6: Tasks in the application

The source files all include a file named "generated_model.h". This is the file that contains the declarations of the components' entry functions and all declarations of non-standard data types for the port variables. It has been generated in a step prior to the synthesis. In the beginning of each task function, the variables that correspond to the inports of the components that are executed in that task are declared. Then follows the code that reproduces the task's control flow by means of components' entry function calls, if-statements and trigger flags. The data flow is implemented by assigning values to the port variables before the components' execution.

`Task0` (Figure 4.7) first executes component `CmpB`, whereupon the first flag for the following joined connection is set. Then, an if-statements checks whether all flags for the joined connection have been set. If the condition is true, component `CmpD` is executed and all flags are reset. Lastly, an if-else-statement carries out the selection connection, which implies the execution of either component `CmpJ` or component `CmpI`, depending the value of component `CmpE`'s outport `outE`. `Task2` contains almost the same code, except for the function call for component `CmpC` instead of `CmpB`. This results from the joined connection which was added to both task trees, due to the mixed trigger types. `Task1` consists of only forked connections and simple connections. The components' entry function are called subsequentially.

```
#include <stdio.h>
#include "C:\save_demo\SAVE\generated_model.h"
#include "globalVars.h"
#include "task0.h"

void task0(void *ignore){    //Trigger: Clock; Period: 20.0
  BOOL trigB_id8;
  BOOL trigD1_id14;
  BOOL trigD2_id15;
  BOOL trigJ_id41;
  BOOL trigI_id39;
  CmpB_id7(trigB_id8, &outB_id9);
  trigD1_id14Trig = 1;
  if(trigD1_id14Trig == 1 && trigD2_id15Trig == 1){
    trigD1_id14 = outB_id9;
    trigD2_id15 = outC_id11;
    CmpD_id13(trigD1_id14, trigD2_id15, &outD_id16);
    trigD1_id14Trig = 0;
    trigD2_id15Trig = 0;
    if(outE_id19>=5){
      trigJ_id41 = outD_id16;
      CmpJ_id40(trigJ_id41);
    }else if(outE_id19<5){
      trigI_id39 = outD_id16;
      CmpI_id38(trigI_id39);
    }
  }
}
```

Figure 4.7: The code implementing `task0`

The configuration file in Figure 4.11 summarizes the tasks' properties. `Task0` and `task1` are periodic tasks with a period of 20. `Task2` is a sporadic task and activated by the signal arriving at external inport `externalTrigger`.

Consequently, the code which implements the SaveCCM design has been successfully generated, which implies that the aspired outcome has been achieved.

```
#include <stdio.h>
#include "C:\save_demo\SAVE\generated_model.h"
#include "globalVars.h"
#include "task1.h"

void task1(void *ignore){    //Trigger: Clock; Period: 20.0
  BOOL trigA_id6;
  InE_id18_combinedtype inE_id18;
  In2_id29_combinedtype in2_id29;
  BOOL in1_id24;
  BOOL trigG_id32;
  In3_id27_combinedtype in3_id27;
  CmpA_id4(trigA_id6, &outA_id5);
  inE_id18.value = outA_id5.value;
  CmpE_id17(inE_id18, &outE_id19);
  in2_id29.value = outA_id5.value;
  Cmp2_id28(in2_id29, &out2_id30);
  in1_id24 = outA_id5;
  Cmp1_id23(in1_id24, &out1_id25);
  trigG_id32 = out2_id30;
  CmpG_id31(trigG_id32);
  in3_id27.value = out1_id25.value;
  Cmp3_id26(in3_id27);
}
```

Figure 4.8: The code implementing `task1`

```
#include <stdio.h>
#include "C:\save_demo\SAVE\generated_model.h"
#include "globalVars.h"
#include "task2.h"

void task2(void *ignore){  //Trigger: external port ->
    externalTrigger#1
  InC_id12_combinedtype inC_id12;
  BOOL trigD1_id14;
  BOOL trigD2_id15;
  BOOL trigJ_id41;
  BOOL trigI_id39;
  CmpC_id10(inC_id12, &outC_id11);
  trigD2_id15Trig = 1;
  if(trigD1_id14Trig == 1 && trigD2_id15Trig == 1){
    trigD1_id14 = outB_id9;
    trigD2_id15 = outC_id11;
    CmpD_id13(trigD1_id14, trigD2_id15, &outD_id16);
    trigD1_id14Trig = 0;
    trigD2_id15Trig = 0;
    if(outE_id19>=5){
      trigJ_id41 = outD_id16;
      CmpJ_id40(trigJ_id41);
    }else if(outE_id19<5){
      trigI_id39 = outD_id16;
      CmpI_id38(trigI_id39);
    }
  }
}
```

Figure 4.9: The code implementing `task2`

```

BOOL outB_id9;
BOOL trigD1_id14Trig;
BOOL outD_id16;
OutA_id5_combinedtype outA_id5;
int outE_id19;
BOOL out2_id30;
Out1_id25_combinedtype out1_id25;
BOOL outC_id11;
BOOL trigD2_id15Trig;
BOOL outD_id16;

```

Figure 4.10: Global outport variables for inter-process communication

```
SYSTEMMODE = NORMAL;
RAM = 65535;

MODE init_mode
{
  RESOLUTION = 10000;

/* Put user-tasks here */
  HARD_TASK PERIODIC TASK0{
    PERIOD_TIME = 20
    OFFSET = 0;
    DEADLINE = 20;
    PRIORITY = 2;
    STACK = 20;
    ROUTINE = task0;
  };

  HARD_TASK PERIODIC TASK1{
    PERIOD_TIME = 20
    OFFSET = 0;
    DEADLINE = 20;
    PRIORITY = 1;
    STACK = 20;
    ROUTINE = task1;
  };

  HARD_TASK APERIODIC TASK2{
    ACTIVATOR = SIGNAL_externalTrigger#1;
    OFFSET = 0;
    DEADLINE = 100;
    PRIORITY = 3;
    STACK = 20;
    ROUTINE = task2;
  };

  SOFT_TASK APERIODIC idle{
    ACTIVATOR = 0;
    OFFSET = 0;
    DEADLINE = 0;
    PRIORITY = 0;
    STACK = 100;
    ROUTINE = idletask;
  };

/* Put waitfree-communication here */

/* default signals for irq, do not edit */

/* Put signals here */
  SIGNAL SIGNAL_externalTrigger#1{
    USER = TASK2;
  }

/* Put semaphores here */

};
```

Figure 4.11: The configuration file for the operating system

4.1.2 Special Test Cases

The general test case, which was investigated in the previous section, showed that all packages of the synthesis deliver the expected results. In this section, a series of special test cases will be briefly examined to show that they are covered by the synthesis as well.

Joined Connection With Only Time Triggers

The example which was investigated in the previous section contains a joined connection with one time trigger and one event trigger, which corresponds to the case of only event triggers. In this section it will be shown that a joined connection with only time triggers is also allocated according to the rule which was introduced in Section 3.1.2.

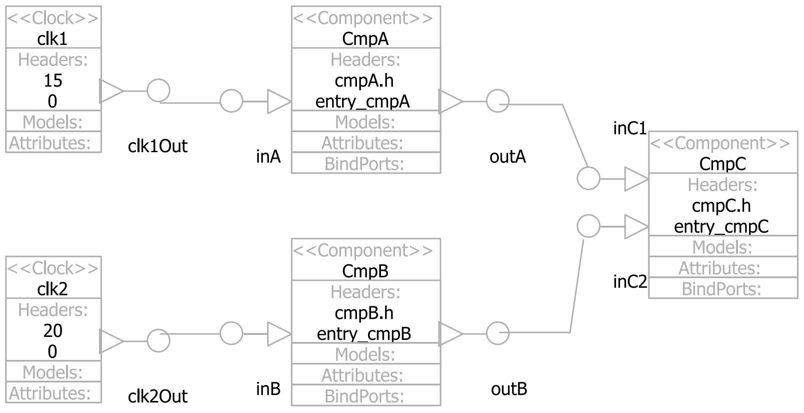


Figure 4.12: Example for a joined connection with only time triggers

Figure 4.12 presents an example for a joined connection with two time triggers. The trigger coming from component `CmpA` has a period of 15, the trigger delivered from component `CmpB` has a period of 20. According to the rule, the joined connection has to be allocated to the task with the longer period. Hence, component `CmpC` has to be allocated to the same task as component `CmpB`. Figure 4.13 presents the two source files that have been generated from the example. As expected, the joined connection, and hence component `CmpC`, is only contained in `task1`, which is the task with the period of 20.

Forked Connection After Selection Connection

This test case will demonstrate, that a forked connection which is attached to a switch is processed properly. The example in Figure 4.14(a) shows such a case. A forked connection leads from the switch's outport `outD2` to the components `CmpC` and `CmpF`. Figure 4.14(b) presents the outcome of the synthesis for the example. As expected, the entry

```
#include <stdio.h>
#include "C:\save_demo\SAVE\generated_model.h"
#include "globalVars.h"
#include "task0.h"

void task0(void *ignore){    //Trigger: Clock; Period: 15.0
  BOOL inA_id6;
  CmpA_id5(inA_id6, &outA_id7);
  inC1_id12Trig = 1;
}
```

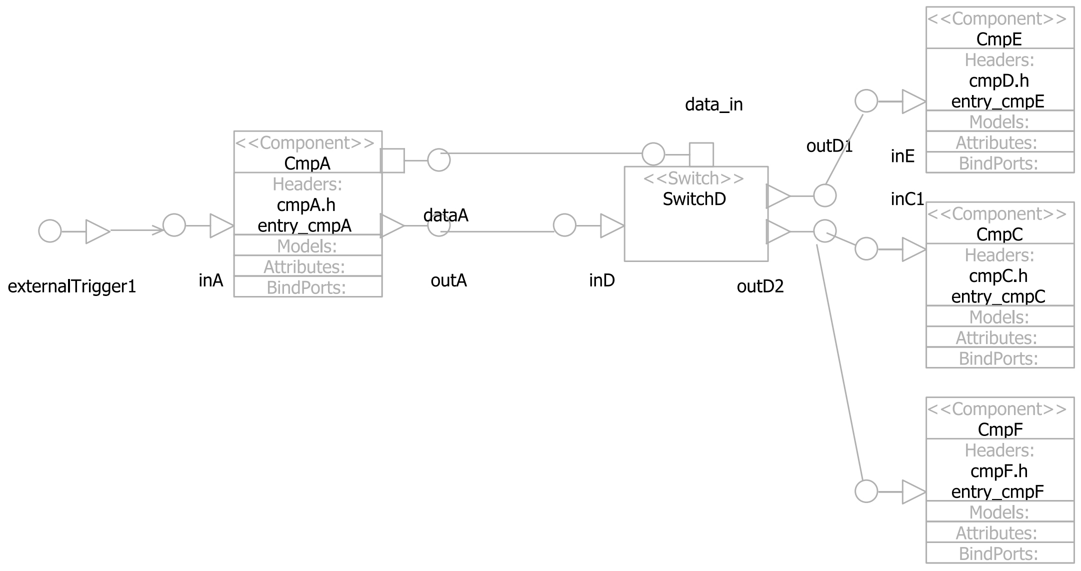
(a) The code for `task0`

```
#include <stdio.h>
#include "C:\save_demo\SAVE\generated_model.h"
#include "globalVars.h"
#include "task1.h"

void task1(void *ignore){    //Trigger: Clock; Period: 20.0
  BOOL inB_id9;
  BOOL inC1_id12;
  BOOL inC2_id13;
  CmpB_id8(inB_id9, &outB_id10);
  inC2_id13Trig = 1;
  if(inC1_id12Trig == 1 && inC2_id13Trig == 1){
    inC1_id12 = outA_id7;
    inC2_id13 = outB_id10;
    CmpC_id11(inC1_id12, inC2_id13);
    inC1_id12Trig = 0;
    inC2_id13Trig = 0;
  }
}
```

(b) The code for `task1`

Figure 4.13: The code implementing the tasks for the joined connection

function calls for components `CmpC` and `CmpF` are both contained in the same if statement block.



(a) The example

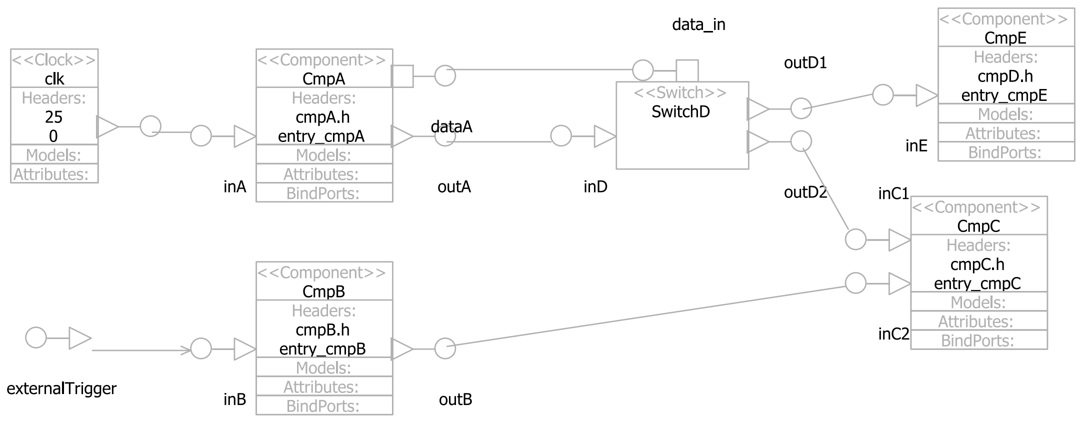
```
#include <stdio.h>
#include "C:\save_demo\SAVE\generated_model.h"
#include "globalVars.h"
#include "task0.h"

void task0(void *ignore){    //Trigger: external port ->
    externalTrigger1#1
  BOOL inA_id3;
  BOOL inC1_id7;
  BOOL _id16;
  BOOL inE_id14;
  CmpA_id2(inA_id3, &outA_id4, &dataA_id5);
  if(dataA_id5>3){
    inC1_id7 = outA_id4;
    CmpC_id6(inC1_id7);
    _id16 = outA_id4;
    CmpF_id15(_id16);
  }else if(dataA_id5<=3){
    inE_id14 = outA_id4;
    CmpE_id13(inE_id14);
  }
}
```

(b) The generated code for the task

Figure 4.14: Example for a forked connection after a selection connection

Joined Connection After Selection Connection

Another interesting case is a joined connection after a selection connection. For the execution of a joined connected component, a trigger flag needs to be set after the execution of each of the trigger delivering components. In the case of a selection connection however, the trigger is not delivered from the preceding component, but by the selection connection. Figure 4.15 shows a corresponding example.



(a) The example

Figure 4.15: Example for a joined connection after a selection connection

Figure 4.16 presents the source code which has been generated for the two tasks of the example. `Task0` contains the selection connection. It can be seen that the flag for the joined connection is set within the if statement block, i.e., right after the condition has been evaluated as being true, the connected inport of the joined connection is triggered.

```
#include <stdio.h>
#include "C:\save_demo\SAVE\generated_model.h"
#include "globalVars.h"
#include "task0.h"

void task0(void *ignore){    //Trigger: Clock; Period: 25.0
  BOOL inA_id3;
  BOOL inC1_id10;
  BOOL inC2_id11;
  BOOL inE_id18;
  CmpA_id2(inA_id3, &outA_id4, &dataA_id5);
  if(dataA_id5 >3){
    inC1_id10Trig = 1;
    if(inC1_id10Trig == 1 && inC2_id11Trig == 1){
      inC1_id10 = outD2_id15;
      inC2_id11 = outB_id8;
      CmpC_id9(inC1_id10, inC2_id11);
      inC1_id10Trig = 0;
      inC2_id11Trig = 0;
    }
  }else if(dataA_id5 <=3){
    inE_id18 = outA_id4;
    CmpE_id17(inE_id18);
  }
}
```

(a) The code for `task0`

```
#include <stdio.h>
#include "C:\save_demo\SAVE\generated_model.h"
#include "globalVars.h"
#include "task1.h"

void task1(void *ignore){    //Trigger: external port ->
    externalTrigger#1
  BOOL inB_id7;
  BOOL inC1_id10;
  BOOL inC2_id11;
  CmpB_id6(inB_id7, &outB_id8);
  inC2_id11Trig = 1;
  if(inC1_id10Trig == 1 && inC2_id11Trig == 1){
    inC1_id10 = outD2_id15;
    inC2_id11 = outB_id8;
    CmpC_id9(inC1_id10, inC2_id11);
    inC1_id10Trig = 0;
    inC2_id11Trig = 0;
  }
}
```

(b) The code for `task1`

Figure 4.16: The generated code for the example in Figure 4.15

4.2 Results

The goal of this thesis was to design and implement an approach to deploy SaveCCM applications. The main objective was the mapping of components to tasks. Predictability was a major requirement for the synthesizing mechanism, hence it needed to be done in a deterministic way.

The previous section showed that the deployment mechanism, which has been designed and implemented within this thesis, fulfills its function and delivers the expected results. Apart from the task source code as final outcome, the synthesis also delivers intermediate results, as for example the task trees, which can be the basis for further analyses.

The presented synthesis has been integrated into the SAVE IDE and will be used as deployment mechanism for the designed applications. Particularly, it will be utilized in a demonstrator project which will exhibit the achievements of the SAVE project.

4.2.1 Use in the SAVE Demonstrator

In the SAVE demonstrator project, an autonomous truck will be utilized to demonstrate how the SAVE IDE supports the development of embedded software. A picture of the truck can be found in Figure 4.17. The truck is equipped with Crossfire MX1 heavy-duty electronic control-units on which the demonstrator application is executed.

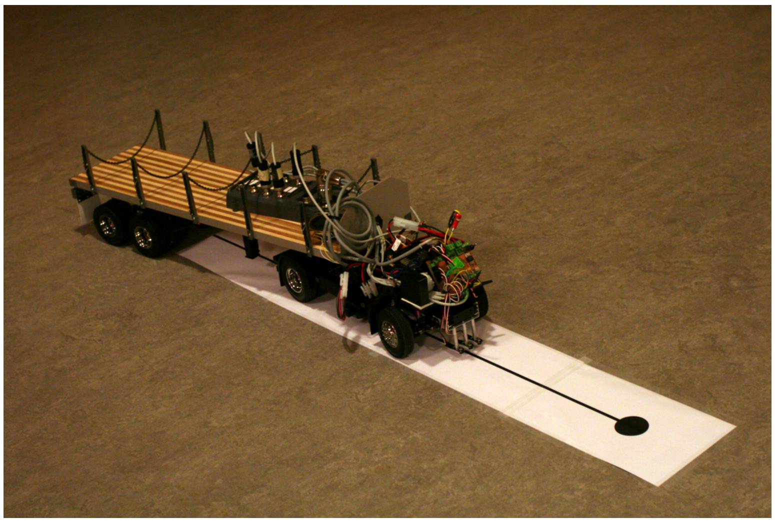


Figure 4.17: Picture of the autonomous truck

Figure 4.18 shows the task which is to be performed by the truck. It is suppossed to follow a straight line and turn around when it reaches its end to then find the line again. A detailed description of the demonstrator application is available in [Sentilles et al., 2009]. The application implementing this task consists of three operational modes, namely:

1. **Follow mode** - to follow the straight line

2. **Turn mode** - to turn at the end of the line

3. **Find mode** - to search the line again after the completed turn

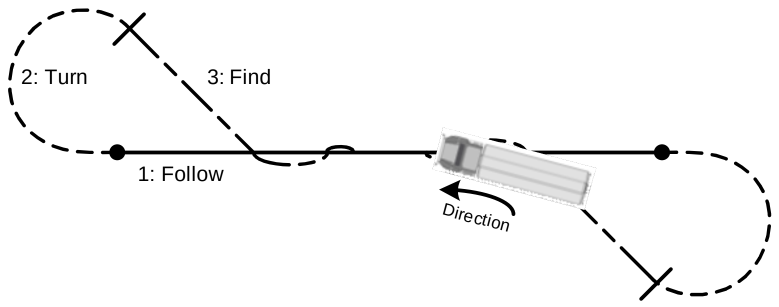


Figure 4.18: The example task which is to be performed by the truck

The corresponding SaveCCM design is presented in Figure 4.19. The `.save` file can be found in Appendix E. Apart from the design tool of the SAVE IDE, this example is also used to demonstrate the tool UPPAAL PORT[2], which is included in the SAVE IDE for modeling, simulation and verification purposes, as well as the synthesis tool, which is subject of this thesis. The result which is achieved when applying the synthesis on the demonstrator application is presented in Appendix F. This generated code is then compiled and executed on the truck demonstrator.

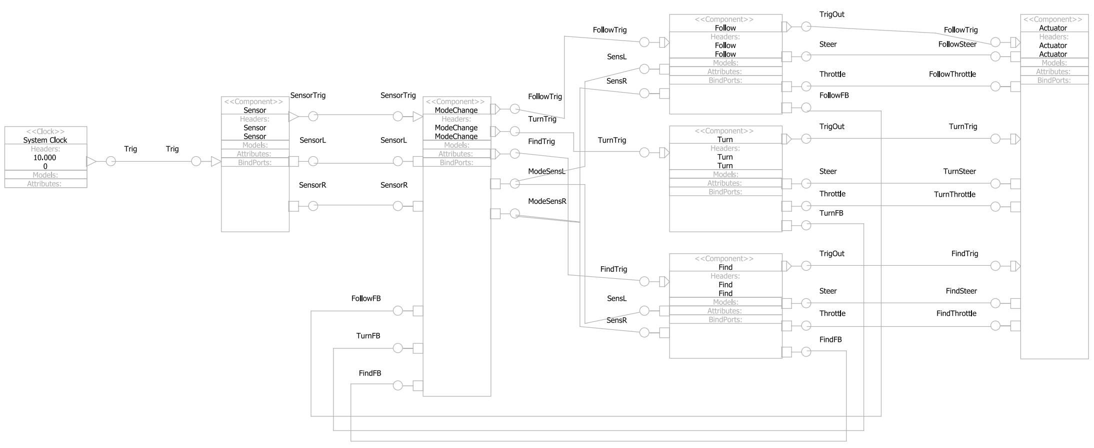


Figure 4.19: The SaveCCM design which implements the example task

[2]see http://www.uppaal.org/port

Chapter 5

Conclusion & Future Work

5.1 Conclusion

This thesis presented an approach for the deployment of SaveCCM designs. The core part of the synthesis mechanism is a set of rules which determine how components are mapped to tasks. The approach has been based on the trigger flow which passes through the component-based designs. A control flow analysis establishes the basis to allocate the components to run-time tasks. Based on the results of the control flow analysis, the allocation rules can be applied, which results in a number of tree structures holding information about the component interconnection structure, referred to as task trees.

The task trees are the basis for the task source code generation. For each task tree, a source code file is generated. During the traversal of a tree, the code which corresponds to each node is added to a task's function. The generation of a configuration file with information about the tasks for the underlying operating system completes the deployment.

By means of a series of test cases, the approach has been proven as applicable. The proposed synthesis mechanism will be used in the SAVE IDE and the further development in the PROGRESS project is prospective.

5.2 Future Work

The synthesis presented in this thesis can be considered a basic first approach for the deployment of SaveCCM designs. It can be improved and further development with respect to various aspects. This section presents some suggestions for future work on the synthesis.

5.2.1 `ProSave`

Since SaveCCM will be superseded by a new, refined component model called ProSave within the PROGRESS research project, the main objective for further development of the synthesis is to adjust it to the new model. ProSave was taken into account throughout design and implementation, hence some preparatory work has already been made. For the full adjustment to the new component model, its schema will be required, which is however to date not yet available. The steps that need to be performed in order to fit the synthesis to the new model will be described in the following.

The main adjustments need to be made in the `parser` package, since it depends on the given schema. The schema for ProSave needs to be available in XML schema for the automatic generation of the ProSave classes with the `xjc` tool and for the automatic unmarshaling of the ProSave design XML files with JAXB. Furthermore, all preprocessing steps that are carried out in the `parser` package have to be adapted to the new ProSave data structures.

Concerning the `controlFlow` package, one additional connection type has to be integrated. With ProSave, the or-connection will be introduced. The `connections` package already contains a class which represents the new connection type, however it is not yet included in the control flow analysis. Thus, the classification of connections needs to be upgraded so that all five connection types are determined correctly. Moreover, the allocation rule for or-connections, as introduced in Section 3.1.2, needs to be added for the construction of task trees.

Finally, the newly introduced connection type also needs to be considered during the code generation. A method which prints the source code corresponding to the or-connection, i.e. a simple call of the component's entry function into each function of the tasks that deliver a trigger to the or-connection, has to be added to the `CodeGenerator` class.

5.2.2 Optimization

The current synthesis represents an approach that has been developed and implemented within the tight time frame of a diploma thesis. The goal to create a mechanism for the automatic deployment of SaveCCM designs has been achieved, yet it could be optimized in several ways. Some of the ideas for optimization, which will be presented in the following, require additional information, which implies extensions to the component model itself.

To achieve a higher level of elaboration, the set of rules that the synthesis is currently based on could be refined. For this purpose, additional parameters could be taken into consideration when deciding about the allocation of components to task. This concerns particularly the case of the and-connection, because its allocation is ambiguous. For the decision making process, other parameters as for instance the current length of the task tree, possibly in terms of its current worst-case execution time (WCET) which can be determined from the chain of already allocated compontents, can be taken into consideration to find the best fitting task. If end-to-end deadlines are provided, the synthesis could attempt to perform the allocation in a way that the sum of the allocated components' WCETs does not exceed the deadlines.

Furthermore, a simplification for the allocation of the and-connection is possible if all incoming triggers originate in the same tasks, especially if it can be assured that all preceding components are executed before the and-connection. In this case the global variables for the incoming triggers of the and-connection are obsolete as well as the if-statement before the call of the and-connected compontent's entryfunction.

The presented synthesis mechanism is designed for applications that are executed on a single shared resource, i.e. all tasks run on the same processor. The deployment of applications for distributed systems would impose additional tasks on the synthesis. Apart from the allocation of components to tasks, it also needs to be dealt with the allocation of components, as well as taks, to physical nodes. Potential isolation rules could define sets of components that are not allowed to be executed on the same node. Vice versa, components could be determined to be executed on the same node or even on a specific node, for example due to physical proximity to a sensor. The issues that a deployment mechanism for distributed systems needs to handle add up to a highly complex problem.

Within another subproject of SAVE and PROGRESS respectively, the connection of real-time databases to the component-based applications is being realized. For the interaction with these databases, a special type of connections will be introduced to connect the application with the database. By means of these connections, data can be read and written from and to the database. To allow for the use of databases, the database access has to be integrated into the source code. All information that is needed for this purpose should be provided by the database connections.

Listings

List of Figures

Bibliography

Åkerholm, M., Carlson, J., Fredriksson, J., Hansson, H., Håkansson, J., Möller, A., Pettersson, P., and Tivoli, M. (2007a). The save approach to component-based development of vehicular systems. *J. Syst. Softw.*, 80(5):655–667. 5, 42

Åkerholm, M., Carlson, J., Håkansson, J., Hansson, H., Nolin, M., Nolte, T., and Pettersson, P. (2007b). The saveccm language reference manual. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-207/2007-1-SE, Mälardalen University. 5, 8, 19

Åkerholm, M., Möller, A., Hansson, H., and Nolin, M. (2005). Towards a dependable component technology for embedded system applications. In *WORDS '05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 320–328, Washington, DC, USA. IEEE Computer Society. 5

Carlson, J., Håkansson, J., and Pettersson, P. (2006). SaveCCM: An analysable component model for real-time systems. In Liu, Z. and Barbosa, L., editors, *Proceedings of the 2nd Workshop on Formal Aspects of Components Software (FACS 2005)*, volume 160 of *Electronic Notes in Theoretical Computer Science*, pages 127–140. Elsevier. 5

Crnkovic, I. (2001). Component-based software engineering - new challenges in software development. *Software Focus*. 4

Fredriksson, J. (2008). *Improving Predictability and Resource Utilization in Component-Based Embedded Real-Time Systems*. PhD thesis, Mälardalen University. 5, 10, 11

Fredriksson, J., Sandström, K., and Åkerholm, M. (2005). Optimizing resource usage in component-based real-time systems. In *the 8th International Symposium on Component-based Software Engineering (CBSE8)*. 2

Genßler, T., Christoph, A., Winter, M., Nierstrasz, O., Ducasse, S., Wuyts, R., Arévalo, G., Schönhage, B., Müller, P., and Stich, C. (2002). Components for embedded software: the pecos approach. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 19–26, New York, NY, USA. ACM. 2

Hänninen, K., Mäki-Turja, J., Nolin, M., Lindberg, M., Lundbäck, J., and Lundbäck, K.-L. (2008). The rubus component model for resource constrained real-time systems. In *3rd IEEE International Symposium on Industrial Embedded Systems*. 2

Kang, B., Kwon, Y.-J., and Lee, R. (2005). A design and test technique for embedded software. *Software Engineering Research, Management and Applications, 2005. Third ACIS International Conference on*, pages 160–165. 4

Ke, X., Sierszecki, K., and Angelov, C. (2007). Comdes-ii: A component-based framework for generative development of distributed real-time control systems. In *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 199–208, Washington, DC, USA. IEEE Computer Society. 2

Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61. 34

Lüders, F. (2004). Adopting a software component model in real-time systems development. In *Proceedings of the 28th Annual NASA/IEEE Software Engineering Workshop*, pages 114–119. IEEE Computer Society Press. 4

Schönhage, B. and van den Born, R. (2002). Model mapping to c++ or java-based ultralight environment. 2

Sentilles, S., Pettersson, A., Nyström, D., Nolte, T., Pettersson, P., and Crnkovic, I. (2009). Save-ide - a tool for design, analysis and implementation of component-based embedded systems. In *Proceedings of the Research Demo Track of the 31st International Conference on Software Engineering (ICSE)*. 42, 60

Stewart, D. B., Volpe, R. A., and Khosla, P. K. (1997). Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Trans. Softw. Eng.*, 23(12):759–776. 2

van Ommering, R., van der Linden, F., Kramer, J., and Magee, J. (2000). The koala component model for consumer electronics software. *Computer*, 33(3):78–85. 8

Appendices

Appendix A

List of Acronyms

CBSE	Component-Based Software Engineering. 4
CPU	central processing unit. 11
RTOS	real-time operating system. 2, 31
SaveCCM	SaveComp Component Model. 2, 5–12, 18, 33, 42–44, 60–62
WCET	worst-case execution time. 31, 64
XML	Extensible Markup Language. 2, 18, 24, 43

Appendix B

Glossary

PROGRESS project
http://www.mrtc.mdh.se/progress/. 5, 11, 12, 19, 62, 63

control flow thread
A part of the application which results in a task tree. Starts with a trigger source (clock or external port).. 13, 19

DTD
Document Type Definition, an XML schema language. 19

Eclipse
A multi-language software development platform comprising an IDE and a plug-in system to extend it (http://www.eclipse.org/).. 42

genetic algorithm
A global search heuristic used to find exact or approximate solutions to optimization and search problems.. 11

JAXB
Java Architecture for XML Binding; an API in the Java EE platform for storing and retrieving data in any XML format. For more information see https://jaxb.dev.java.net/. 18, 19, 63

ProSave
The component model introduced in the PROGRESS project.. 63

simulated annealing
A generic probabilistic metaheuristic for locating a good approximation to the global minimum of a given function in a large search space.. 11

task tree
The tree representation of a task. Generated during the control flow analysis as a result of the application of the allocation rules.. 24, 25, 31, 32, 34, 47, 51, 60, 62–64

unmarshalling
The process of retrieving the memory representation of an object from a data format suitable for storage or transmission, as for example from XML to Java objects. 19

XML schema
An XML schema language, recommended by W3C.. 19, 63

Appendix C

The SaveCCM XML Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!-- strings als ids-->
<xsd:element name="APPLICATION" type="Application"/>
<xsd:element name="comment" type="xsd:string"/>
<xsd:complexType name="Application">
  <xsd:sequence>
    <xsd:element name="IODEF" type="IODef"/>
    <xsd:element name="TYPEDEFS" type="TypeDefs"/>
  <xsd:element name="COMPONENTLIST" type="ComponentList"/>
  <xsd:element name="CONNECTIONLIST" type="ConnectionList"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="IODef">
  <xsd:sequence>
    <xsd:element name="INPORT" type="Inport" minOccurs="0" maxOccurs="
        unbounded"/>
    <xsd:element name="OUTPORT" type="Outport" minOccurs="0" maxOccurs=
        "unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="TypeDefs">
  <xsd:sequence>
    <xsd:element name="COMPONENTDESC" type="ComponentDesc" minOccurs="0
        " maxOccurs="unbounded"/>
    <xsd:element name="SWITCHDESC" type="SwitchDesc" minOccurs="0"
        maxOccurs="unbounded"/>
  <xsd:element name="ASSEMBLYDESC" type="AssemblyDesc" minOccurs="0"
      maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

```
<xsd:complexType name="ComponentDesc">
  <xsd:sequence>
    <xsd:element name="INPORT" type="Inport" minOccurs="0" maxOccurs="
        unbounded"/>
    <xsd:element name="OUTPORT" type="Outport" minOccurs="0" maxOccurs=
        "unbounded"/>
  <xsd:element name="ATTRIBUTE" type="Attribute" minOccurs="0"
      maxOccurs="unbounded"/>
  <xsd:element name="BEHAVIOUR" type="Behaviour"/>
  <xsd:element name="REALISATION" type="Realisation"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="Realisation">
  <xsd:choice>
    <xsd:element name="ENTRYFUNC" type="EntryFunc"/>
      <xsd:element name="CLOCK" type="Clock"/>
    <xsd:element name="DELAY" type="Delay"/>
    <xsd:sequence>
      <xsd:element name="COMPONENTLIST" type="ComponentList"/>
      <xsd:element name="CONNECTIONLIST" type="ConnectionList"/>
    </xsd:sequence>
  </xsd:choice>
</xsd:complexType>

<xsd:complexType name="Clock">
  <xsd:attribute name="period" type="xsd:string" use="required"/>
  <xsd:attribute name="jitter" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="Delay">
  <xsd:attribute name="delay" type="xsd:string" use="required"/>
  <xsd:attribute name="precision" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="SwitchDesc">
  <xsd:sequence>
    <xsd:element name="INPORT" type="Inport" minOccurs="0" maxOccurs="
        unbounded"/>
    <xsd:element name="OUTPORT" type="Outport" minOccurs="0" maxOccurs=
        "unbounded"/>
  <xsd:element name="SWITCHCONDITION" type="SwitchCondition" minOccurs=
      "0" maxOccurs="unbounded"/>
  </xsd:sequence>
<xsd:attribute name="id" type="xsd:string" use="required"/>
</xsd:complexType>
```

```
<xsd:complexType name="SwitchCondition">
  <xsd:sequence>
    <xsd:element name="FROM" type="From"/>
  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="TO" type="To"/>
    <xsd:element name="CONDITION" type="Condition"/>
  </xsd:sequence>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Condition">
  <xsd:attribute name="setport" type="xsd:string" use="required"/>
  <xsd:attribute name="value" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="AssemblyDesc">
  <xsd:sequence>
    <xsd:element name="INPORT" type="Inport" minOccurs="0" maxOccurs="
        unbounded"/>
    <xsd:element name="OUTPORT" type="Outport" minOccurs="0" maxOccurs=
        "unbounded"/>
  <xsd:element name="COMPONENTLIST" type="ComponentList"/>
  <xsd:element name="CONNECTIONLIST" type="ConnectionList"/>
  </xsd:sequence>
<xsd:attribute name="id" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="ComponentList">
  <xsd:sequence>
      <xsd:element name="COMPONENT" type="Component" minOccurs="0"
          maxOccurs="unbounded"/>
      <xsd:element name="SWITCH" type="Switch" minOccurs="0" maxOccurs=
          "unbounded"/>
    <xsd:element name="ASSEMBLY" type="Assembly" minOccurs="0"
        maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ConnectionList">
  <xsd:sequence>
    <xsd:element name="CONNECTION" type="Connection" minOccurs="0"
        maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Component">
  <xsd:attribute name="type" type="xsd:string" use="required"/>
```

```
    <xsd:attribute name="id" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="Switch">
    <xsd:attribute name="type" type="xsd:string" use="required"/>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="Assembly">
    <xsd:attribute name="type" type="xsd:string" use="required"/>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="Behaviour">
    <xsd:sequence>
      <xsd:element name="MODEL" type="Model" minOccurs="0" maxOccurs="
          unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Model">
    <xsd:sequence>
      <xsd:element ref="comment" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="filename" type="xsd:string"/>
    <xsd:attribute name="type" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="EntryFunc">
    <xsd:sequence>
      <xsd:element name="BINDPORT" type="Bindport" minOccurs="0"
          maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="filename" type="xsd:string" use="required"/>
    <xsd:attribute name="entry" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="Bindport">
    <xsd:attribute name="port" type="xsd:string" use="required"/>
    <xsd:attribute name="argument" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="Inport">
    <xsd:attribute name="mode" type="Mode" use="required"/>
    <xsd:attribute name="type" type="xsd:string" use="required"/>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
    <xsd:attribute name="value" type="xsd:string"/>
    <xsd:attribute name="external" type="xsd:string"/>
    <xsd:attribute name="setport" type="Setport" default="false"/>
  </xsd:complexType>

  <xsd:complexType name="Outport">
    <xsd:attribute name="mode" type="Mode" use="required"/>
    <xsd:attribute name="type" type="xsd:string" use="required"/>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
    <xsd:attribute name="value" type="xsd:string"/>
    <xsd:attribute name="external" type="xsd:string"/>
  </xsd:complexType>

  <xsd:simpleType name="Mode">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="data"/>
      <xsd:enumeration value="trig"/>
      <xsd:enumeration value="combined"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="Setport">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="true"/>
      <xsd:enumeration value="false"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType name="Attribute">
    <xsd:attribute name="id" type="xsd:string" use="required"/>
    <xsd:attribute name="type" type="xsd:string" use="required"/>
    <xsd:attribute name="value" type="xsd:string" use="required"/>
    <xsd:attribute name="credibility" type="xsd:string"/>
  </xsd:complexType>

  <xsd:complexType name="Connection">
    <xsd:sequence>
      <xsd:element name="FROM" type="From"/>
    <xsd:element name="TO" type="To" minOccurs="0" maxOccurs="unbounded"/
       >
    <xsd:element name="BEHAVIOUR" type="Behaviour" minOccurs="0"
       maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="From">
    <xsd:attribute name="id" type="xsd:string" use="required"/>
    <xsd:attribute name="port" type="xsd:string" use="required"/>
  </xsd:complexType>
```

```
<xsd:complexType name="To">
  <xsd:attribute name="id" type="xsd:string" use="required"/>
  <xsd:attribute name="port" type="xsd:string" use="required"/>
</xsd:complexType>

</xsd:schema>
```

Appendix D

`.save` File for the General Test Case

```
<APPLICATION id="example1">

  <IODEF>
    <INPORT mode="combined" type="int" id="externalTrigger#1" external=
        "true" setport="false"/>
  </IODEF>

  <TYPEDEFS>
    <COMPONENTDESC id="clk#2">
      <OUTPORT mode="trig" type="void" id="clk_out#3"/>
      <REALISATION>
        <CLOCK period="20" jitter="0"/>
      </REALISATION>
    </COMPONENTDESC>

    <COMPONENTDESC id="CmpA#4">
      <INPORT mode="trig" type="void" id="trigA#6"/>
      <OUTPORT mode="combined" type="int" id="outA#5"/>
      <REALISATION>
        <ENTRYFUNC filename="cmpA.h" entry="entryCmpA">
        </ENTRYFUNC>
      </REALISATION>
    </COMPONENTDESC>

    <COMPONENTDESC id="CmpB#7">
      <INPORT mode="trig" type="void" id="trigB#8"/>
      <OUTPORT mode="trig" type="void" id="outB#9"/>
      <REALISATION>
        <ENTRYFUNC filename="cmpB.h" entry="entryCmpB">
        </ENTRYFUNC>
      </REALISATION>
    </COMPONENTDESC>
```

```
<COMPONENTDESC id="CmpC#10">
  <INPORT mode="combined" type="int" id="inC#12" setport="false"/>
  <OUTPORT mode="trig" type="void" id="outC#11"/>
  <REALISATION>
    <ENTRYFUNC filename="cmpC.h" entry="entryCmpC">
    </ENTRYFUNC>
  </REALISATION>
</COMPONENTDESC>

<COMPONENTDESC id="CmpD#13">
  <INPORT mode="trig" type="void" id="trigD1#14"/>
  <INPORT mode="trig" type="void" id="trigD2#15"/>
  <OUTPORT mode="trig" type="void" id="outD#16"/>
  <REALISATION>
    <ENTRYFUNC filename="cmpD.h" entry="entryCmpD">
    </ENTRYFUNC>
  </REALISATION>
</COMPONENTDESC>

<COMPONENTDESC id="CmpE#17">
  <INPORT mode="combined" type="int" id="inE#18" setport="false"/>
  <OUTPORT mode="data" type="int" id="outE#19"/>
  <REALISATION>
    <ENTRYFUNC filename="cmpE.h" entry="entryCmpE">
    </ENTRYFUNC>
  </REALISATION>
</COMPONENTDESC>

<COMPONENTDESC id="Cmp1#23">
  <INPORT mode="trig" type="void" id="in1#24"/>
  <OUTPORT mode="combined" type="int" id="out1#25"/>
  <REALISATION>
    <ENTRYFUNC filename="cmp1.h" entry="entryCmp1">
    </ENTRYFUNC>
  </REALISATION>
</COMPONENTDESC>

<COMPONENTDESC id="Cmp3#26">
  <INPORT mode="combined" type="int" id="in3#27" setport="false"/>
  <REALISATION>
    <ENTRYFUNC filename="cmp3.h" entry="entryCmp3">
    </ENTRYFUNC>
  </REALISATION>
</COMPONENTDESC>

<COMPONENTDESC id="Cmp2#28">
  <INPORT mode="combined" type="int" id="in2#29" setport="false"/>
  <OUTPORT mode="trig" type="void" id="out2#30"/>
```

```
    <REALISATION>
      <ENTRYFUNC filename="cmp2.h" entry="entryCmp2">
      </ENTRYFUNC>
    </REALISATION>
  </COMPONENTDESC>

  <COMPONENTDESC id="CmpG#31">
    <INPORT mode="trig" type="void" id="trigG#32"/>
    <REALISATION>
      <ENTRYFUNC filename="cmpG.h" entry="entryCmpG">
      </ENTRYFUNC>
    </REALISATION>
  </COMPONENTDESC>

  <COMPONENTDESC id="CmpI#38">
    <INPORT mode="trig" type="void" id="trigI#39"/>
    <REALISATION>
      <ENTRYFUNC filename="cmpI.h" entry="entryCmpI">
      </ENTRYFUNC>
    </REALISATION>
  </COMPONENTDESC>

  <COMPONENTDESC id="CmpJ#40">
    <INPORT mode="trig" type="void" id="trigJ#41"/>
    <REALISATION>
      <ENTRYFUNC filename="cmpJ.h" entry="entryCmpJ">
      </ENTRYFUNC>
    </REALISATION>
  </COMPONENTDESC>

  <SWITCHDESC id="SwitchH#33">
    <INPORT mode="trig" type="void" id="inH#34"/>
    <INPORT mode="data" type="int" id="a#37" setport="true"/>
    <OUTPORT mode="trig" type="void" id="outH1#35"/>
    <OUTPORT mode="trig" type="void" id="outH2#36"/>
    <SWITCHCONDITION>
      <FROM id="SwitchH#33" port="inH#34"/>
      <TO id="SwitchH#33" port="outH2#36"/>
      <CONDITION setport="a#37" value="t%gt;=5"/>

      <TO id="SwitchH#33" port="outH1#35"/>
      <CONDITION setport="a#37" value="t%lt;5"/>
    </SWITCHCONDITION>
  </SWITCHDESC>

  <ASSEMBLYDESC id="AssemblyF#20">
    <INPORT mode="combined" type="int" id="inF#21" setport="false"/>
    <OUTPORT mode="trig" type="void" id="outF#22"/>
```

```
      <COMPONENTLIST>
        <COMPONENT type="Cmp1#23" id="Cmp1#23Impl"/>
        <COMPONENT type="Cmp3#26" id="Cmp3#26Impl"/>
        <COMPONENT type="Cmp2#28" id="Cmp2#28Impl"/>
      </COMPONENTLIST>
      <CONNECTIONLIST>
        <CONNECTION>
          <FROM id="AssemblyF#20Impl" port="outF#22"/>
          <TO id="CmpG#31Impl" port="trigG#32"/>
        </CONNECTION>
        <CONNECTION>
          <FROM id="Cmp1#23Impl" port="out1#25"/>
          <TO id="Cmp3#26Impl" port="in3#27"/>
        </CONNECTION>
        <CONNECTION>
          <FROM id="AssemblyF#20Impl" port="inF#21"/>
          <TO id="Cmp2#28Impl" port="in2#29"/>
          <TO id="Cmp1#23Impl" port="in1#24"/>
        </CONNECTION>
        <CONNECTION>
          <FROM id="Cmp2#28Impl" port="out2#30"/>
          <TO id="AssemblyF#20Impl" port="outF#22"/>
        </CONNECTION>
      </CONNECTIONLIST>
    </ASSEMBLYDESC>
  </TYPEDEFS>

  <COMPONENTLIST>
    <COMPONENT type="CmpA#4" id="CmpA#4Impl"/>
    <COMPONENT type="CmpB#7" id="CmpB#7Impl"/>
    <COMPONENT type="CmpC#10" id="CmpC#10Impl"/>
    <COMPONENT type="CmpD#13" id="CmpD#13Impl"/>
    <COMPONENT type="CmpE#17" id="CmpE#17Impl"/>
    <COMPONENT type="CmpG#31" id="CmpG#31Impl"/>
    <COMPONENT type="CmpI#38" id="CmpI#38Impl"/>
    <COMPONENT type="CmpJ#40" id="CmpJ#40Impl"/>
    <COMPONENT type="clk#2" id="clk#2Impl"/>
    <SWITCH type="SwitchH#33" id="SwitchH#33Impl"/>
    <ASSEMBLY type="AssemblyF#20" id="AssemblyF#20Impl"/>
  </COMPONENTLIST>

  <CONNECTIONLIST>
    <CONNECTION>
      <FROM id="clk#2Impl" port="clk_out#3"/>
      <TO id="CmpB#7Impl" port="trigB#8"/>
      <TO id="CmpA#4Impl" port="trigA#6"/>
    </CONNECTION>
    <CONNECTION>
```

```
      <FROM id="CmpC#10Impl" port="outC#11"/>
      <TO id="CmpD#13Impl" port="trigD2#15"/>
    </CONNECTION>
    <CONNECTION>
      <FROM id="CmpB#7Impl" port="outB#9"/>
      <TO id="CmpD#13Impl" port="trigD1#14"/>
    </CONNECTION>
    <CONNECTION>
      <FROM id="CmpD#13Impl" port="outD#16"/>
      <TO id="SwitchH#33Impl" port="inH#34"/>
    </CONNECTION>
    <CONNECTION>
      <FROM id="SwitchH#33Impl" port="outH1#35"/>
      <TO id="CmpI#38Impl" port="trigI#39"/>
    </CONNECTION>
    <CONNECTION>
      <FROM id="SwitchH#33Impl" port="outH2#36"/>
      <TO id="CmpJ#40Impl" port="trigJ#41"/>
    </CONNECTION>
    <CONNECTION>
      <FROM id="CmpE#17Impl" port="outE#19"/>
      <TO id="SwitchH#33Impl" port="a#37"/>
    </CONNECTION>
    <CONNECTION>
      <FROM id="example1#0Impl" port="externalTrigger#1"/>
      <TO id="CmpC#10Impl" port="inC#12"/>
    </CONNECTION>
    <CONNECTION>
      <FROM id="CmpA#4Impl" port="outA#5"/>
      <TO id="CmpE#17Impl" port="inE#18"/>
      <TO id="AssemblyF#20Impl" port="inF#21"/>
    </CONNECTION>
  </CONNECTIONLIST>
</APPLICATION>
```

Appendix E

.save File for the Demonstrator Project

```
<APPLICATION id="DEMOAPP">

  <IODEF>

  </IODEF>

  <TYPEDEFS>
    <COMPONENTDESC id="Sensor#1">
      <INPORT mode="trig" type="void" id="Trig#5"/>
      <OUTPORT mode="trig" type="void" id="SensorTrig#2"/>
      <OUTPORT mode="data" type="int" id="SensorL#3"/>
      <OUTPORT mode="data" type="int" id="SensorR#4"/>

      <REALISATION>
        <ENTRYFUNC filename="Sensor" entry="Sensor">
        </ENTRYFUNC>
      </REALISATION>
    </COMPONENTDESC>

    <COMPONENTDESC id="ModeChange#6">
      <INPORT mode="trig" type="void" id="SensorTrig#7"/>
      <INPORT mode="data" type="int" id="SensorL#8" setport="false"/>
      <INPORT mode="data" type="int" id="SensorR#9" setport="false"/>
      <INPORT mode="data" type="int" id="FollowFB#15" setport="false"/>
      <INPORT mode="data" type="int" id="TurnFB#16" setport="false"/>
      <INPORT mode="data" type="int" id="FindFB#17" setport="false"/>
      <OUTPORT mode="data" type="int" id="ModeSensL#10"/>
      <OUTPORT mode="data" type="int" id="ModeSensR#11"/>
      <OUTPORT mode="combined" type="int" id="FollowTrig#12"/>
      <OUTPORT mode="combined" type="int" id="TurnTrig#13"/>
      <OUTPORT mode="combined" type="int" id="FindTrig#14"/>

      <REALISATION>
        <ENTRYFUNC filename="ModeChange" entry="ModeChange">
        </ENTRYFUNC>
      </REALISATION>
    </COMPONENTDESC>

    <COMPONENTDESC id="Follow#18">
      <INPORT mode="combined" type="int" id="FollowTrig#19" setport="
          false"/>
      <INPORT mode="data" type="int" id="SensL#20" setport="false"/>
      <INPORT mode="data" type="int" id="SensR#21" setport="false"/>
      <OUTPORT mode="data" type="int" id="FollowFB#22"/>
      <OUTPORT mode="combined" type="int" id="TrigOut#23"/>
      <OUTPORT mode="data" type="int" id="Steer#24"/>
      <OUTPORT mode="data" type="int" id="Throttle#25"/>

      <REALISATION>
        <ENTRYFUNC filename="Follow" entry="Follow">

        </ENTRYFUNC>
      </REALISATION>
    </COMPONENTDESC>

    <COMPONENTDESC id="Turn#26">
      <INPORT mode="combined" type="int" id="TurnTrig#27" setport="
          false"/>
      <OUTPORT mode="data" type="int" id="TurnFB#28"/>
      <OUTPORT mode="data" type="int" id="Steer#29"/>
      <OUTPORT mode="data" type="int" id="Throttle#30"/>
      <OUTPORT mode="combined" type="int" id="TrigOut#31"/>

      <REALISATION>
        <ENTRYFUNC filename="Turn" entry="Turn">
        </ENTRYFUNC>
      </REALISATION>
    </COMPONENTDESC>

    <COMPONENTDESC id="Find#32">
      <INPORT mode="combined" type="int" id="FindTrig#33" setport="
          false"/>
      <INPORT mode="data" type="int" id="SensL#34" setport="false"/>
      <INPORT mode="data" type="int" id="SensR#35" setport="false"/>
      <OUTPORT mode="data" type="int" id="FindFB#36"/>
      <OUTPORT mode="combined" type="int" id="TrigOut#37"/>
      <OUTPORT mode="data" type="int" id="Steer#38"/>
      <OUTPORT mode="data" type="int" id="Throttle#39"/>
      <REALISATION>
        <ENTRYFUNC filename="Find" entry="Find">
        </ENTRYFUNC>
      </REALISATION>
    </COMPONENTDESC>

    <COMPONENTDESC id="Actuator#40">
      <INPORT mode="combined" type="int" id="FollowTrig#41" setport="
          false"/>
      <INPORT mode="combined" type="int" id="TurnTrig#42" setport="
          false"/>
      <INPORT mode="combined" type="int" id="FindTrig#43" setport="
          false"/>
      <INPORT mode="data" type="int" id="FollowSteer#44" setport="false
          "/>
      <INPORT mode="data" type="int" id="FollowThrottle#45" setport="
          false"/>
      <INPORT mode="data" type="int" id="TurnSteer#46" setport="false"/
          >
      <INPORT mode="data" type="int" id="TurnThrottle#47" setport="
          false"/>
      <INPORT mode="data" type="int" id="FindSteer#48" setport="false"/
          >
      <INPORT mode="data" type="int" id="FindThrottle#49" setport="
          false"/>

      <REALISATION>
        <ENTRYFUNC filename="Actuator" entry="Actuator">
        </ENTRYFUNC>
      </REALISATION>
    </COMPONENTDESC>

    <COMPONENTDESC id="System Clock#50">
      <OUTPORT mode="trig" type="void" id="Trig#51"/>

      <REALISATION>
        <CLOCK period="10000" jitter="0"/>
      </REALISATION>
    </COMPONENTDESC>

</TYPEDEFS>

<COMPONENTLIST>
  <COMPONENT type="Sensor#1" id="Sensor#1Impl"/>
  <COMPONENT type="ModeChange#6" id="ModeChange#6Impl"/>
  <COMPONENT type="Follow#18" id="Follow#18Impl"/>
  <COMPONENT type="Turn#26" id="Turn#26Impl"/>
    <COMPONENT type="Find#32" id="Find#32Impl"/>
    <COMPONENT type="Actuator#40" id="Actuator#40Impl"/>
    <COMPONENT type="System Clock#50" id="System Clock#50Impl"/>
</COMPONENTLIST>

<CONNECTIONLIST>
  <CONNECTION>
    <FROM id="Sensor#1Impl" port="SensorTrig#2"/>
    <TO id="ModeChange#6Impl" port="SensorTrig#7"/>
  </CONNECTION>

  <CONNECTION>
    <FROM id="Sensor#1Impl" port="SensorL#3"/>
    <TO id="ModeChange#6Impl" port="SensorL#8"/>
  </CONNECTION>

  <CONNECTION>
    <FROM id="Sensor#1Impl" port="SensorR#4"/>
    <TO id="ModeChange#6Impl" port="SensorR#9"/>
  </CONNECTION>

  <CONNECTION>
    <FROM id="ModeChange#6Impl" port="FollowTrig#12"/>
    <TO id="Follow#18Impl" port="FollowTrig#19"/>
  </CONNECTION>

  <CONNECTION>
    <FROM id="ModeChange#6Impl" port="TurnTrig#13"/>
    <TO id="Turn#26Impl" port="TurnTrig#27"/>
  </CONNECTION>

  <CONNECTION>
    <FROM id="ModeChange#6Impl" port="FindTrig#14"/>
    <TO id="Find#32Impl" port="FindTrig#33"/>
  </CONNECTION>

  <CONNECTION>
    <FROM id="ModeChange#6Impl" port="ModeSensL#10"/>
    <TO id="Follow#18Impl" port="SensL#20"/>
    <TO id="Find#32Impl" port="SensL#34"/>
  </CONNECTION>

  <CONNECTION>
    <FROM id="ModeChange#6Impl" port="ModeSensR#11"/>
    <TO id="Follow#18Impl" port="SensR#21"/>
    <TO id="Find#32Impl" port="SensR#35"/>
  </CONNECTION>
```

```
<CONNECTION>
  <FROM id="Find#32Impl" port="FindFB#36"/>
  <TO id="ModeChange#6Impl" port="FindFB#17"/>
</CONNECTION>

<CONNECTION>
  <FROM id="Turn#26Impl" port="TurnFB#28"/>
  <TO id="ModeChange#6Impl" port="TurnFB#16"/>
</CONNECTION>

<CONNECTION>
  <FROM id="Follow#18Impl" port="FollowFB#22"/>
  <TO id="ModeChange#6Impl" port="FollowFB#15"/>
</CONNECTION>

<CONNECTION>
  <FROM id="Follow#18Impl" port="TrigOut#23"/>
  <TO id="Actuator#40Impl" port="FollowTrig#41"/>
</CONNECTION>

<CONNECTION>
  <FROM id="Turn#26Impl" port="TrigOut#31"/>
  <TO id="Actuator#40Impl" port="TurnTrig#42"/>
</CONNECTION>

<CONNECTION>
  <FROM id="Find#32Impl" port="TrigOut#37"/>
  <TO id="Actuator#40Impl" port="FindTrig#43"/>
</CONNECTION>

<CONNECTION>
  <FROM id="Follow#18Impl" port="Steer#24"/>
  <TO id="Actuator#40Impl" port="FollowSteer#44"/>
</CONNECTION>

<CONNECTION>
  <FROM id="Follow#18Impl" port="Throttle#25"/>
  <TO id="Actuator#40Impl" port="FollowThrottle#45"/>
</CONNECTION>

<CONNECTION>
  <FROM id="Turn#26Impl" port="Steer#29"/>
  <TO id="Actuator#40Impl" port="TurnSteer#46"/>
</CONNECTION>

<CONNECTION>
  <FROM id="Turn#26Impl" port="Throttle#30"/>
  <TO id="Actuator#40Impl" port="TurnThrottle#47"/>
```

```
    </CONNECTION>

    <CONNECTION>
      <FROM id="Find#32Impl" port="Throttle#39"/>
      <TO id="Actuator#40Impl" port="FindThrottle#49"/>
    </CONNECTION>

    <CONNECTION>
      <FROM id="Find#32Impl" port="Steer#38"/>
      <TO id="Actuator#40Impl" port="FindSteer#48"/>
    </CONNECTION>

    <CONNECTION>
      <FROM id="System Clock#50Impl" port="Trig#51"/>
      <TO id="Sensor#1Impl" port="Trig#5"/>
    </CONNECTION>
  </CONNECTIONLIST>

</APPLICATION>
```

Appendix F

Generated Task Code for the Demonstrator Project

```
#include <stdio.h>
#include "C:\save_demo\SAVE\generated_model.h"
#include "globalVars.h"
#include "task0.h"

void task0(void *ignore){    //Trigger: Clock; Period: 10000.0
  BOOL Trig_id5;
  BOOL SensorTrig_id7;
  int SensorL_id8;
  int SensorR_id9;
  int FollowFB_id15;
  int TurnFB_id16;
  int FindFB_id17;
  FollowTrig_id19_combinedtype FollowTrig_id19;
  int SensL_id20;
  int SensR_id21;
  TurnTrig_id27_combinedtype TurnTrig_id27;
  FindTrig_id33_combinedtype FindTrig_id33;
  int SensL_id34;
  int SensR_id35;
  FollowTrig_id41_combinedtype FollowTrig_id41;
  TurnTrig_id42_combinedtype TurnTrig_id42;
  FindTrig_id43_combinedtype FindTrig_id43;
  int FollowSteer_id44;
  int FollowThrottle_id45;
  int TurnSteer_id46;
  int TurnThrottle_id47;
  int FindSteer_id48;
  int FindThrottle_id49;
  Sensor_id1(Trig_id5, &SensorTrig_id2, &SensorL_id3, &SensorR_id4);
  SensorTrig_id7 = SensorTrig_id2;
  SensorL_id8 = SensorL_id3;
  SensorR_id9 = SensorR_id4;
  FollowFB_id15 = FollowFB_id22;
  TurnFB_id16 = TurnFB_id28;
  FindFB_id17 = FindFB_id36;
  ModeChange_id6(SensorTrig_id7, SensorL_id8, SensorR_id9,
      FollowFB_id15, TurnFB_id16, FindFB_id17, &ModeSensL_id10, &
      ModeSensR_id11, &FollowTrig_id12, &TurnTrig_id13, &FindTrig_id14)
      ;
  FollowTrig_id19.value = FollowTrig_id12.value;
  SensL_id20 = ModeSensL_id10;
  SensR_id21 = ModeSensR_id11;
  Follow_id18(FollowTrig_id19, SensL_id20, SensR_id21, &FollowFB_id22,
      &TrigOut_id23, &Steer_id24, &Throttle_id25);
  FollowTrig_id41Trig = 1;
  TurnTrig_id27.value = TurnTrig_id13.value;
  Turn_id26(TurnTrig_id27, &TurnFB_id28, &Steer_id29, &Throttle_id30, &
      TrigOut_id31);
  TurnTrig_id42Trig = 1;
  FindTrig_id33.value = FindTrig_id14.value;
  SensL_id34 = ModeSensL_id10;
  SensR_id35 = ModeSensR_id11;
  Find_id32(FindTrig_id33, SensL_id34, SensR_id35, &FindFB_id36, &
      TrigOut_id37, &Steer_id38, &Throttle_id39);
  FindTrig_id43Trig = 1;
  if(FollowTrig_id41Trig == 1 && TurnTrig_id42Trig == 1 &&
      FindTrig_id43Trig == 1){
    FollowTrig_id41.value = TrigOut_id23.value;
    TurnTrig_id42.value = TrigOut_id31.value;
    FindTrig_id43.value = TrigOut_id37.value;
    FollowSteer_id44 = Steer_id24;
    FollowThrottle_id45 = Throttle_id25;
    TurnSteer_id46 = Steer_id29;
    TurnThrottle_id47 = Throttle_id30;
    FindSteer_id48 = Steer_id38;
    FindThrottle_id49 = Throttle_id39;
    Actuator_id40(FollowTrig_id41, TurnTrig_id42, FindTrig_id43,
        FollowSteer_id44, FollowThrottle_id45, TurnSteer_id46,
        TurnThrottle_id47, FindSteer_id48, FindThrottle_id49);
    FollowTrig_id41Trig = 0;
    TurnTrig_id42Trig = 0;
    FindTrig_id43Trig = 0;
  }
}
```